UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA EN INFORMÁTICA

PROYECTO FIN DE CARRERA

SIMULACIÓN DE ENTORNOS VIRTUALES EN 3D E IMPLEMENTACIÓN DE AGENTES CON EMOCIONES BÁSICAS

Autor: Reiji Rodrigo Yamazaki Martínez

Director: Prof. Javier Carbó Rubiera

Año 2016

AGRADECIMIENTOS

Quiero agradecer a mis familiares, mis amigos y a mis compañeros de trabajo por todo el apoyo y ánimo que me han ido dando año tras año, para poder terminar este proyecto. También quiero dar las gracias a mi tutor Javier por todos sus consejos.

Índice

Índice de ilustraciones

Índice de tablas

CAPÍTULO 1: **Introducción**

Este capítulo se describirá primero el contexto en el que nos encontramos, después el enfoque del trabajo realizado. Y finalmente, se explicarán brevemente el resto de capítulos de este documento.

1.1 CONTEXTO

Desde la aparición pública del término Inteligencia Artificial (IA) en 1956 [1], hasta comienzos de los 80, en las investigaciones en este campo había una tendencia a seguir un formalismo común que ahora se suele denominar como Good Old Fashioned Artificial Intelligence (GOFAI) [1] [2]. En esta época los estudios se centraban en el procesamiento de símbolos y era frecuente encontrarse con investigaciones sobre sistemas expertos y algoritmos de búsquedas donde se centraba en el procesamiento de la información de la razón y la lógica de los seres humanos. A principios de los 70, Herbert Simon y Allen Newell propusieron la hipótesis de los sistemas de símbolos físicos (Physical Symbol System Hypothesis) [3] que dice lo siguiente: "todas las cosas del mundo se pueden representar mediante símbolos, y la inteligencia se puede considerar como operaciones de estos símbolos" [4]. Con el tiempo se empezaron a ver las limitaciones del GOFAI, y a partir de 1980, empiezan a surgir investigaciones en las que consideran como esencia de la inteligencia la realización de acciones en el mundo real y la adaptación a los entornos no deterministas. El límite de la IA basada en procesamiento de símbolos fue señalado en 1986 por Rodney Brooks [4]. Brooks y sus compañeros, al ver que con estas técnicas era difícil construir robots que se desenvuelvan razonablemente bien en el mundo real, ideó la robótica conductista centrada en la arquitectura subsunción. Esta arquitectura poseía un sistema de respuestas simples a los estímulos recibidos por los distintos sensores y fue posible crear robots que interactúan con un entorno real y con movimientos más ágiles que cuando se utilizaban las técnicas tradicionales. Sin embargo, Masahiro Fujita, uno de los creadores de AIBO y QRIO de SONY [4], comenta que diseñaron estos robots basándose en esta arquitectura, pero por limitaciones técnicas en el sistema de aprendizaje, hubo que pre-programar muchos comportamientos. Su objetivo actual es diseñar un robot que sea capaz de hacer emerger de forma ordenada las funciones que necesita al ir interactuando con el entorno. Le da importancia a la materialización del cuerpo (embodiment) para poder desarrollar la inteligencia [4].

Desde el punto de vista ideológico, en la actualidad existen dos posturas diferentes basados en los objetivos que se desea lograr con el desarrollo de la inteligencia artificial. La primera postura es la IA fuerte: postura de los que tienen como objetivo dilucidar por

completo los mecanismos de nuestra inteligencia y crear robots inteligentes que actúan con los mismos mecanismos que nosotros. Y la segunda postura es la IA débil: postura de aquellos que no les importan los mecanismos (métodos) con los que se ejecute la actividad intelectual, sino que se le da importancia al artefacto inteligente (software) que se haya construido [2].

Estas dos posturas anteriores no son incompatibles, de hecho, la mayoría de los investigadores aceptan y utilizan las dos posturas en sus trabajos, aunque normalmente se tenga una orientación más fuerte hacia una de ellas que la otra.

Con los mecanismos de GOFAI es posible conseguir programas que sean más inteligentes que las personas en un ámbito concreto. Sin embargo, hay entornos en los que es más difícil explotar sus capacidades sólo con procesamiento de símbolos, como es en el mundo real. Este escenario es un escenario donde las IA basadas en procesamiento de símbolos son débiles. Una comparativa que se usa muy a menudo es, que los programas creados con este método pueden ganar al campeón del ajedrez del mundo, sin embargo, no puede por sí sólo colocar las piezas ni desplazarse a la habitación de al lado para coger el tablero [2].

Desde el punto de vista de la optimización de recursos, las técnicas de GOFAI desarrolla su gran potencial. Pero en la actualidad también existen otros puntos de vista para abordar los problemas de optimización. Son métodos que, en vez de utilizar directamente los símbolos, utilizan el concepto de “individuo”. Cada vez hay más investigaciones que se centran en los comportamientos de los seres vivos para utilizarlos como método para encontrar la solución óptima. El proceso de adaptación de los grupos de seres vivos al entorno se puede considerar como un tipo de algoritmo de optimización. Los individuos se adaptan al entorno a través del aprendizaje, la especie a través de la evolución, y los grupos a través de la cooperación entre los individuos que los compone [5].

Como método de IA aplicable en el mundo real, la tendencia actual, como ya se comentó anteriormente con la arquitectura de subsunción, es crear agentes autónomos que son reactivos, centrando el enfoque en la interacción con el entorno, implementando actos reflejos con movimientos simples que permite adaptarse a un entorno no determinista con facilidad. Este enfoque se suele llamar como “New AI” siendo un modelo más apto para interaccionar con el mundo real que con las técnicas de GOFAI que, desde este punto de vista, se considera que no es suficiente para adaptarse al mundo real [2].

1.2 ENFOQUE

En este PFC se ha enfocado la materialización del cuerpo para intentar simular los mecanismos más instintivos y primitivos de nuestra inteligencia mediante las emociones

básicas (por lo tanto, con mayor inclinación hacia a la postura de IA fuerte). Para ello se ha utilizado el modelo de estados emocionales PAD propuesto por Albert Mehrabian y James A. Russell en 1974 [6] [7]. Para permitir evaluar los resultados de una forma intuitiva, se ha pensado en crear algún sistema que permita observar a los individuos de forma visual. Lo ideal sería construir un robot humanoide que sea capaz de manifestar sus emociones básicas al ir interaccionando con el entorno. Sin embargo, por razones obvias (falta de recursos: tiempo, dinero, y personal cualificado), se ha abordado el problema creando un mundo virtual en tres dimensiones. Dentro de este mundo virtual se lanzarán varios agentes autónomos, con sus propios objetivos a cumplir. Dotar de objetivos a cada agente favorece la percepción de distintos tipos de estímulos en sus organismos. Esto es fundamental para que sus estados internos del organismo varíen y, por lo tanto, se manifiesten las emociones hacia el exterior. Esto es una imitación muy simplificada de nuestras vidas: el hecho de hacer "cosas" e interaccionar con otras personas nos hace sentir distintos tipos de emociones, mucho más variadas y ricas que si estuviésemos encerrados en nuestra habitación sin hacer nada. Y encontramos una semejanza a esta tesis en los comentarios de Fujita que dice lo siguiente: “los robots autónomos no se mueven sin motivación” [4].

1.3 CONTENIDO

En el capítulo 2, se detallará el estado de la cuestión. Se hablará de las distintas formas de crear un mundo virtual y de los mecanismos descubiertos hasta ahora sobre la generación de las emociones humanas, y sus aplicaciones en agentes.

En el capítulo 3, se contará cómo se ha ido desarrollando este proyecto, detallando los recursos utilizados, los pasos seguidos, y sus costes.

En el capítulo 4, se detallan los objetivos que se desean alcanzar con la realización de este trabajo.

En el capítulo 5, se hablará de la arquitectura del motor de simulación y de los detalles de implementación de sus componentes principales.

En el capítulo 6, se describirán los detalles de implementación de los agentes y su modelo de estados emocionales utilizado en el contexto del mundo de cajas.

En el capítulo 7, se detallarán los resultados del experimento final. En este capítulo se determinará cómo y en qué situaciones se han manifestado las emociones en los agentes, analizando el proceso seguido por cada agente para evaluar si es posible dar una explicación lógica a sus actos.

En el capítulo 8, se comentarán las conclusiones obtenidas con la realización de este trabajo.

En el capítulo 9, se enumerarán las posibles mejoras que se podrían realizar sobre este trabajo en el futuro.

Y en el último capítulo: el capítulo 10, se detallarán todos los documentos, libros y sitios webs consultados para la elaboración de este trabajo.

Se ha añadido, como anexos, un resumen de la evolución histórica de la tecnología 3D para ordenadores, y otra sección que detalla paso por paso la creación del agente utilizado en este proyecto para aquellos que quieran conocer estos aspectos en más detalle.

CAPÍTULO 2: **Estado de la cuestión**

En este capítulo, se exponen brevemente las herramientas y tecnologías que existen actualmente para crear un avatar en un mundo virtual en 3D. Posteriormente se explicará lo que son las emociones básicas y el funcionamiento del cerebro humano relacionado con la manifestación de éstas. Y también se hablará sobre un trabajo sobre las emociones humanas aplicado en un humano virtual, y que ha sido la base para elaborar el presente PFC.

2.1 SECOND LIFE

Second Life [8] es un entorno virtual en 3D desarrollado y mantenido por Linden Lab, ha tenido mayor acogida sobre todo por EE.UU. Se trata de un mundo virtual online, donde sus habitantes pueden construir y poseer sus propiedades. Cada uno de nosotros podemos tener una representación virtual dentro de este mundo en forma de avatar. Con nuestro avatar podemos movernos dentro del mundo virtual, cambiarle la apariencia, vivir en nuestra propiedad o comunicarnos con otros avatares.

La forma de conseguir nuestro avatar es sencilla: registrarse en la página web de Second Life. Cuando uno se registra como usuario, se le asigna un avatar, que podemos personalizar desde el primer instante en el que entramos en el mundo de Second Life.

Una vez dentro, nuestro avatar puede interactuar con cualquier elemento del entorno mediante envío de mensajes.

Para un usuario avanzado, Second Life nos da la posibilidad de programar nuestros entornos mediante un lenguaje de scripting propietario de Linden Labs: el LSL (Linden Scripting Language). La API de LSL tiene todas las funciones necesarias para manipular las entidades virtuales en Second Life.

Engloban funciones para manipular vectores, enviar mensajes, sistema de detección de colisiones, creación de figuras geométricas básicas, manipulación de vehículos, etc.

Como entorno de desarrollo de LSL existen diversos programas fáciles de obtener a través de internet:

- **LSL-Editor** [9]: permite compilar y depurar tu código LSL.
- **LSL-Plus** [10]**:** es un plugin del IDE Eclipse, y funciona como analizador sintáctico y depurador de LSL.

El problema que tienen estos editores es que sólo permiten depurar sin renderizar el entorno 3D, por lo que dificulta la depuración de funciones que manipulen vectores y entidades en 3D.

Para ejecutar tu propio script dentro de un entorno 3D hay que instalar el Second Life Viewer [11] y conectarse a los servidores de Linden Lab. Una vez dentro de Second Life, el Second Life Viewer nos permite cargar nuestro script LSL y ejecutarlo en él. Estas son las ventajas e inconvenientes que existen al utilizar este entorno:

Ventajas:

- LSL se puede desarrollar en cualquier entorno: "*Write once, debug everywhere"*.
- Una API completa para manipular los avatares y otros objetos 3D.

Inconvenientes:

- Hace falta conectarse a internet para poder visualizar el entorno 3D.
- Hace falta registrarse en la página de Second Life.
- Para crear una segunda cuenta o para utilizar opciones avanzadas hay que pagar.
- No existen IDE que permitan depurar levantando un entorno de simulación en 3D.

2.2 JGOMAS

JGomas [12] es un sistema multi-agente sobre Jade orientado a juegos. Fue desarrollado en la Universidad Politécnica de Valencia, con el propósito de estudiar la integración entre un sistema multi-agente (MAS) y un sistema de realidad virtual. En este MAS está implementado el juego de captura de bandera, que consiste en enfrentar a dos bandos: Aliados y Eje, alrededor de una bandera. Los agentes Aliados deben ir a la base del Eje para capturar la bandera, y llevarla a su base. Por otro lado, el Eje tiene el deber de proteger su bandera. Para ello, en cuanto su bandera sea capturada por un agente Aliado, deberá recuperarla y devolverla en su lugar. Cada partida está determinada por un tiempo máximo que dispondrá el bando Aliado para conseguir su objetivo. Cada agente perteneciente a los dos bandos desenvuelve un rol concreto que puede ser:

- **Soldier**: combate con el bando contrario y da apoyo a sus aliados.
- **Medic**: acude a curar a los agentes heridos de su bando.
- **FieldOps**: acude para proporcionar munición a sus aliados.

La aplicación está compuesta, principalmente, de dos subsistemas. El primero es un sistema multi-agente, implementado en JADE, donde se encuentra la lógica del juego y los bandos enfrentados. El segundo es un Visualizador Gráfico encargado de renderizar el estado del juego en tres dimensiones que ha sido desarrollado en C++ como un módulo externo y no como un agente. Para el desarrollo del visualizador se ha utilizado una herramienta llamada OpenSceneGraph que funciona sobre OpenGL. Uno de los agentes del MAS actúa como interfaz del Visualizador y envía el estado del campo de batalla a través de sockets al

Visualizador Gráfico para que éste dibuje, en tres dimensiones, los agentes, muros y la bandera.

Los agentes son totalmente personalizables e incluso es posible incluir nuevos roles en el sistema para aumentar la complejidad del juego. Todos los detalles técnicos necesarios para ampliar el juego están detallados en el manual de usuario de JGomas. No sólo los agentes son ampliables, también es posible personalizar el campo de batalla, de forma fácil, mediante un fichero de texto. En estos ficheros de texto se señalan mediante asteriscos las casillas donde queramos situar los obstáculos. Una vez cargado el fichero en la aplicación, el visualizador dibuja el escenario con los obstáculos que estén definidos en el fichero.

2.3 HSP+EASY3D

HSP es un lenguaje de programación basado en BASIC, comenzó a desarrollarse en el año 1994 y publicado, de forma gratuita, a partir de 1996 [13]. Este lenguaje tiene una muy buena aceptación por los programadores amateurs por su sencillez. Su uso es muy extendido en Japón para programar videojuegos y programas de utilidad de diversos ámbitos. La licencia es totalmente gratuita y se puede utilizar incluso para fines comerciales sin ningún coste.

Easy3D es un plugin para HSP que permite manejar y renderizar entornos tridimensionales. Para utilizar este plugin basta con tener unas nociones básicas de cálculos con vectores en tres dimensiones. Además de tener unos comandos para crear geometrías básicas y realizar transformaciones en 3D, es capaz de detectar las colisiones entre los modelos 3D, y permite también cargar modelos 3D y animaciones diseñadas en otras herramientas externas de diseño 3D.

Easy3D funciona bajo DirectX, por lo que su uso será posible sólo para Windows al igual que el intérprete de HSP.

2.4 MAXSCRIPT

Es un lenguaje de scripting desarrollado únicamente para su uso bajo la aplicación 3D Studio Max [14]. El 3D Studio Max es una aplicación que permite diseñar y animar gráficos en tres dimensiones. MaxScript [15] fue creado para que los usuarios del 3D Studio pudieran automatizar las tareas repetitivas o ampliar funcionalidades de la aplicación. Su API permite crear nuevas ventanas dentro de la aplicación que permita ejecutar nuestros scripts, además de permitir manejar los objetos tridimensionales en el editor. La ejecución es sencilla, podemos introducir el script a mano mediante la consola de MaxScript o cargando un fichero externo que contiene nuestro código.

La creación de un avatar mediante MaxScript no parece imposible, pero sus limitaciones no son pocas. Para poder ejecutar nuestro código tenemos que tener instalado en nuestro ordenador el 3D Studio Max con su licencia correspondiente que no compensa comprarla si sólo lo queremos utilizar para ejecutar nuestro avatar. Tampoco existen librerías, desarrolladas para simular mundos virtuales en 3D con MaxScript, como por ejemplo un motor físico para detectar colisiones y simular la dinámica de objetos del mundo real. Esto no significa que esto no sea posible, pero sería necesario implementarlo desde cero y para ello hay que tener unos conocimientos amplios sobre el tema.

2.5 DIRECTX

Es un conjunto de API desarrolladas por Microsoft, y utilizadas para la creación de contenido multimedia. Está compuesto por varios componentes, y cada uno tiene una determinada responsabilidad. Posee un componente específico para gestionar las entradas del usuario (teclado, ratón, mandos de juegos, etc.), otro componente encargado de reproducir audio y/o video, otro para gestionar las comunicaciones de red, y otro para dibujar imágenes tridimensionales, entre otros [16]. El componente encargado de dibujar las imágenes en 3D se llama Direct3D, y ésta es la porción más utilizada de DirectX en los videojuegos.

Como se ha mencionado anteriormente, DirectX se emplea en los desarrollos de videojuegos para ordenadores domésticos. Y el mercado de los videojuegos ha ido evolucionando en los últimos veinte años de una forma vertiginosa, creando cada vez más consumidores de estas tecnologías. Para llegar a esto, ha sido importante la evolución de los dispositivos físicos del ordenador, concretamente de las tarjetas aceleradoras de gráficos o tarjetas gráficas. Dada la popularidad de los videojuegos en los ordenadores personales, las capacidades de las tarjetas gráficas han ido mejorando con las nuevas versiones de DirectX y Windows.

2.6 OPENGL

Desarrollado por Silicon Graphics Inc en 1992. OpenGL viene de Open Graphic Library. Define una API estandarizada que permite acceder al hardware gráfico y es independiente de la plataforma y el lenguaje de programación. La palabra “Open” no viene de open source sino por no depender de ningún sistema operativo en concreto, es sólo una interfaz software. Los sistemas operativos que deseen soportar OpenGL se licenciarán e implementarán una versión propia de OpenGL siguiendo su API. Pero también existe una implementación no oficial de OpenGL como es el caso de Mesa 3D, una librería gráfica open source que utiliza

la misma estructura y sintaxis que OpenGL, pero sin poseer la licencia. A diferencia de DirectX, OpenGL permite a los fabricantes definir sus propias extensiones y así dotar de nuevas funcionalidades a sus tarjetas gráficas. Muchas de estas extensiones son revisadas por varios fabricantes y son incluidas en posteriores versiones de OpenGL, cosa que con DirectX no ocurre ya que la totalidad de sus API las gestiona Microsoft. Otra diferencia a destacar es que todas las versiones de OpenGL son compatibles con sus versiones anteriores, cuando esto no ocurre con las versiones de DirectX (aunque en las últimas versiones se está consiguiendo un cierto nivel de retrocompatibilidad) [17].

2.7 MOTORES FÍSICOS

Una API gráfica sólo se encarga de generar una imagen bidimensional a partir de modelos en 2D o 3D. Sin embargo, para representar un modelo del mundo real hacen falta las leyes físicas para detectar las colisiones entre cuerpos, y para simular los cuerpos rígidos o blandos en movimiento.

Un motor físico lleva implementados los algoritmos necesarios para simular los modelos físicos de la mecánica clásica. Cabe destacar que existen motores con gran precisión de cálculo utilizados en investigaciones científicas o fines académicos para simular entornos reales. Pero también, existen otros menos precisos para sistemas que requieren una solución inmediata como en los videojuegos. Los motores físicos más conocidos son Havok [18], Bullet [19] y ODE [20].

2.8 ENTORNO VIRTUAL EN 3D

Para la realización de un entorno virtual en 3D, se necesita al menos un **modelo visible** (una malla) y un **simulador de entorno**. El simulador se encargará de que los modelos visibles recobren "vida" dentro de la pantalla del ordenador.

2.8.1 VISIÓN GENERAL DEL MODELADO

Estos son los pasos simplificados para poder crear, al menos de forma austera, la figura del avatar en 3D. Primero de todo hace falta modelar su malla mediante polígonos. Una vez terminado, se recomienda aplicar unas texturas para definir los colores que tendrá el avatar en cada parte de su cuerpo. Terminada y aplicada la textura ya estaría terminada la parte correspondiente a la definición de la apariencia del modelo visible.

Sin embargo, para que este modelo recobre vida, es necesario definir una serie de movimientos con el detalle que se consideren necesarios para que se desenvuelva en el entorno objetivo. Estos pueden ser: caminar, correr, saltar, agacharse, tumbarse, o cualquier

otro movimiento que podamos imaginar. Es esencial seleccionar cuidadosamente estos movimientos para que estén acordes con el contexto y el entorno en el que se va a desenvolver éste. Una vez terminado el avatar con su textura y movimientos ya definidos, el siguiente paso es diseñar un escenario por donde éste pueda moverse.

2.8.2 FASES DEL PROCESO DE MODELADO

En este apartado se van a detallar más en profundidad cada uno de los pasos que se suele seguir habitualmente durante el proceso de elaboración de un modelo 3D.

1. **Modelado de la malla:** Para el modelado del avatar, se recomienda utilizar una aplicación diseñada y creada para tal fin. Se podría utilizar un programa muy conocido como el 3D Studio Max, o Blender [21] que no es tan conocido como el primero, pero ofrece multitud de funcionalidades como el 3D Studio. Este segundo es, a diferencia del primero, gratuito bajo los términos de la licencia GPL y cada vez va ganando más popularidad. En este caso vamos a utilizar este último, ya que existe una amplia documentación con tutoriales en su página oficial, y en varios idiomas sin ningún coste.

2. **Diseño de las texturas:** Para dibujar las texturas es necesario mapear cada vértice de la malla sobre una superficie en dos dimensiones. Esta transformación se realiza utilizando una técnica llamada "*UV mapping*" [22]. Esto es posible realizar tanto con 3D Studio Max como con Blender. Una vez creado el mapa UV se exporta a una imagen para poder dibujar las texturas con alguna herramienta de dibujo. En este proyecto utilizaremos una herramienta llamada GIMP [23].

3. **Animación de la malla:** Para crear los movimientos del avatar es posible utilizar el 3D Studio Max o el Blender al igual que en la fase de modelado de la malla. Ambos permiten animar mediante FK o IK creando un esqueleto para la malla. FK es la abreviatura de *Forward Kinematic* [22], algo así como, "Cinemática hacia delante". Es una técnica de animación de modelos 3D que cuando movemos una parte del esqueleto, calcula la nueva posición de los huesos adyacentes hasta llegar a los extremos. Y la técnica IK que viene de *Inverse Kinematic* [22] y puede ser algo así como "Cinemática inversa". En IK se realizan los cálculos de los huesos adyacentes hacia el tronco del esqueleto. Es útil cuando queremos mover una extremidad hacia otro punto del espacio, ya que, con sólo mover la extremidad el algoritmo se encarga de calcular las posiciones de los huesos adyacentes "hacia dentro" justo al contrario

que en FK. La IK es más compleja de realizar incluso con el 3D Studio o Blender. Sin embargo, cuando se consigue configurar bien los parámetros, los resultados obtenidos son asombrosos pudiendo representar unos movimientos muy realistas.

Para el modelado del escenario no es necesario seguir la fase de animación ya que será un cuerpo estático. Como herramienta para diseñar un escenario se puede utilizar también el 3D Studio Max o Blender. Pero si lo que se desea es diseñar un escenario natural con montañas, lagos y rocas, se puede utilizar el Terragen [24] que permite generar paisajes mediante una serie de algoritmos estadísticos.

En el anexo de este documento se ha añadido un caso práctico realizado con Blender donde se describen los pasos seguidos para modelar el avatar utilizado en este proyecto.

2.9 EMOCIONES

Cuando escuchamos hablar de emociones podemos pensar en multitud de términos, algunos simples, y otros más complejos y difíciles de entender. Sin embargo, en este trabajo se va a centrar en un tipo muy concreto y fundamental de las emociones: las emociones básicas.

2.9.1 EMOCIONES BÁSICAS

Durante gran parte del siglo XX, la mayoría de los antropólogos culturales defendían el hecho de que las emociones se transmitían a través de la cultura. Sin embargo, el experimento realizado a finales de los 60, por el psicólogo norteamericano Paul Ekman desmiente parte de esta teoría. El experimento, que se realizó para corroborar la teoría cultural, hizo evidenciar lo contrario: la existencia de una serie de emociones que son comunes entre distintas culturas; y Ekman las llamó “emociones básicas”.

Darwin vio similitud con los animales. Dijo que las emociones innatas son comunes entre personas y en muchos animales por las similitudes en sus expresiones [25].

En la aplicación a crear, cada agente será capaz de sentir y expresar un total de cinco emociones básicas. Estas emociones, han sido seleccionadas desde las emociones básicas que define Paul Ekman, y las emociones básicas que definió Robert Plutchik en su Rueda de las Emociones [26]. Debido a que cada autor define un conjunto de emociones básicas distintas, se ha decidido implementar las emociones básicas que tienen en común que son las siguientes:

- Alegría

- Tristeza
- Ira
- Miedo
- Sorpresa

Según Ekman, las emociones básicas son una serie de reacciones que todo ser humano tiene de forma innata [27]. Antes de su experimento, se pensaba que una emoción era "algo aprendida", de la cultura en la que uno vive, al igual que el idioma. Sin embargo, con su experimento Ekman demostró que había una serie de emociones que eran comunes para personas que han crecido en culturas totalmente distintas como la cultura norteamericana y la cultura de la tribu Fore de Papúa Nueva Guinea. Ekman centró sus investigaciones en las expresiones faciales asociadas a estas emociones. Las expresiones faciales son básicas para comunicarnos entre personas, y su interpretación es común en todo el mundo sea del país que sea, y de la cultura que sea. Estas son las emociones básicas que definió Ekman como resultado del experimento:

- Alegría
- Tristeza
- Ira
- Miedo
- Sorpresa
- Aversión

En cambio, Robert Plutchik define las emociones básicas como: biológicamente primitivas y que las sienten todos los animales, no sólo los humanos [28]. Sirven como mecanismo para aumentar la probabilidad de supervivencia de las especies animales. En la Rueda de las Emociones que definió Plutchik enumeran cuatro pares de emociones básicas, donde cada par son emociones opuestas. Estos son los pares:

- Alegría ↔ Tristeza
- Confianza ↔ Aversión
- Miedo ↔ Ira
- Sorpresa ↔ Anticipación

Según Dylan Evans [27], las emociones básicas se manifiestan de forma repentina y tiene un tiempo relativamente corto de duración, apenas unos segundos.

Nótese que se ha omitido la implementación de la emoción de aversión que se nombra por los dos investigadores. Esta emoción se produce cuando hay rechazo hacia algo o alguien. Se ha tomado la decisión de no incluirla en la lista porque en el escenario que se va a crear, no hay comida para alimentarse ni olores que son los elementos que pensamos que habitualmente producen esta emoción.

2.9.2 MODELO DE ESTADOS EMOCIONALES

Para la generación de las emociones del agente se ha basado en el modelo de estado emocional PAD [29]. Donde PAD viene de las siglas: Pleasure, Arousal y Dominance. En 1974, Albert Mehrabian y James A. Russell descubrió en sus experimentos que los estados emocionales de las personas se pueden representar mediante la combinación de estas tres dimensiones. Cada sigla representa una dimensión en el espacio.

El estado emocional del agente estará representado por un punto (P, A, D) dentro del espacio de este modelo. En este espacio tridimensional, se definen varias regiones que indica la probabilidad de que el agente se dé cuenta de que está experimentando una experiencia emocional; cuanto más cerca esté el punto del centro de la región de una determinada emoción, más probabilidad habrá de que el agente sienta dicha emoción.

2.9.3 LAS EMOCIONES Y EL SUBCONSCIENTE

Al parecer existe una parte de nuestro subconsciente que nos hace sentir una sensación "buena" o "mala" ante determinados estímulos sin que la consciencia intervenga en la interpretación de dicho estímulo. Damasio y LeDoux corroboran esto a través de sus experimentos [30] [31]. Y tiene relación con lo que dijo William James; que el hombre no huye al encontrarse con un oso porque tiene miedo, sino que, al estar huyendo siente el miedo [25]. Se puede interpretar que existe ese "algo" que nos hace huir, que nos haya empujado a tomar esa acción, y posteriormente notamos que el corazón late muy deprisa, la sangre fluye más rápido por todo el cuerpo y tenemos tensión en los músculos. Y el hecho de darnos cuenta de estos cambios repentinos en el interior de nuestro cuerpo hace emerger la emoción del miedo. Esta es la teoría que más seguidores tiene en la actualidad. Cuando hasta el siglo pasado se pensaba que era la emoción la que nos hacía tomar la decisión de huir.

William James formuló la teoría de que los estímulos que activan las emociones son capaces de producir respuestas corporales acorde con la emoción activada. La sensación de sentir una emoción se produce cuando la consciencia reconoce la activación de la respuesta corporal originada por un estímulo emocional. Pensó que cada emoción produce una

reacción corporal distinta y eso nos hace sentir una emoción distinta por cada tipo de sensación percibida [25].

LeDoux afirma que la emoción se puede definir como un proceso del cerebro para calcular o valorar un estímulo. Primero ocurre la respuesta emocional (reacción corporal y cambios fisiológicos), y posteriormente el cerebro decide que hay "algo" importante y nosotros nos damos cuenta de que estamos reaccionando a ese "algo" y surgen los sentimientos (la interpretación consciente del estímulo emocional) [31].

Damasio afirma que las emociones proporcionan un mecanismo para que el cerebro y la mente puedan evaluar el entorno de fuera y el interior del propio organismo, y producir una respuesta adecuada. Las emociones y sus respuestas están asociadas con el cuerpo, y los sentimientos están relacionados con la mente. Las emociones generan respuestas simples que favorece la supervivencia del organismo y es anterior al sentimiento [32].

La emoción es un mecanismo regulador del organismo que se ha ido refinando con el tiempo y se ha ido transmitiendo a través de los genes. Otra función de las emociones es que sirve para preparar el organismo para responder al estímulo que ha captado el subconsciente [30]. Cuando el cerebro detecta una amenaza, libera adrenalina en el flujo sanguíneo, incrementando la respiración y el ritmo cardíaco, haciéndonos sudar. Esto prepara el organismo para huir corriendo de la amenaza o enfrentarse y luchar contra ella [33].

Las amenazas no tienen por qué ser reconocidas por la consciencia para producir una respuesta defensiva. El consciente reconoce la amenaza e interpreta la emoción asociando con el objeto que originó la amenaza después de que la parte no consciente haya realizado un primer proceso de filtrado de los estímulos [34].

También existe un mecanismo interno del organismo que está estrechamente relacionado con el proceso de generación de las emociones. Esto es la homeostasis. Es un mecanismo para mantener el organismo en un nivel de equilibrio correcto [30]. Es un mecanismo que auto-regula la vida, y nos induce a un estado no sólo de vida sino a un estado mejor de felicidad [32].

Existe otro mecanismo importante que posee la parte no consciente de nuestro cerebro, y es la de orientar nuestra atención hacia algo. Nuestra atención es selectiva, el cerebro focaliza en cosas que son relevantes en cada momento e ignora el resto. De no ser así estaríamos saturados procesando cada cosa que percibimos. El cerebro va aprendiendo a seleccionar las cosas importantes con el tiempo, de forma inconsciente [33]. Cuando prestamos atención a algo y nos concentramos, determinados circuitos cerebrales se activan y el hecho de estar activo, potencia aún más nuestra capacidad de concentración sobre el contexto. Esto permite

mantener la atención un mayor tiempo sobre algo que nuestro cerebro considera importante [31].

A continuación, se enumeran, a modo de resumen, los principales componentes del cerebro que intervienen durante el procesamiento y/o generación de los estímulos emocionales y algunas de sus funciones:

- **Memoria de trabajo**: La memoria de trabajo sólo permite realizar una única tarea a la vez. Por lo que cualquier evento que nos haga distraer y consiga sobrescribir nuestra memoria de trabajo, nos hará olvidar momentáneamente lo que estábamos realizando hasta hace un momento. La memoria de trabajo es fundamental para el pensamiento y la resolución de problemas. La memoria de trabajo contiene sólo la información de lo que estamos prestando atención, pero también tiene acceso a la memoria de larga duración para permitir deliberar las siguientes acciones, ampliando la probabilidad de encontrar la mejor solución [31].

- **Memoria emocional**: Los sucesos y las entidades centrales que nos hace sentir alguna emoción, son más susceptibles de perdurar en nuestra memoria que otras que no nos hace sentir nada [27]. Recordar una señal emocional del pasado nos hace centrar la atención en la parte más importante del problema a resolver, y consciente o inconscientemente, ayuda a mejorar la precisión de la solución a buscar. Marca las experiencias buenas y malas del pasado para utilizar como heurística en las futuras búsquedas de soluciones. Damasio lo llama Marcador Somático [32]. Damasio también habla del "As if body loop", que es un mecanismo que tiene el cerebro para "simular" las actividades internas que el organismo manifestó a través de los estímulos que se percibió en el pasado, y es capaz de reproducir sus efectos como si estuviésemos percibiendo los estímulos de verdad [30]. La memoria emocional puede ser reactivada cuando se recuerda y puede ser modificada. Esto se denomina ***reconsolidación*** y tiene una ventana de tiempo de unas 6 horas en la cual se puede alterar la memoria [33]. Cuando vivimos experiencias con emociones, el cerebro lo considera como un hecho importante y lo almacena en nuestra memoria explícita (memoria accesible desde el consciente) para poder recuperarlo desde la memoria de trabajo más adelante. Los recuerdos explícitos son más fáciles de recordar, si el estado emocional del ser humano coincide, con el estado emocional en el momento de memorizar el hecho. Por ejemplo, si nos sentimos deprimidos actualmente, es más fácil recordar los hechos tristes que los felices.

- **Amígdalas**: sirven para aprender sobre qué sentimos miedo. Activan nuestra atención sobre algo importante, siendo capaz de anular la tarea actual e introducir en la memoria de

trabajo la información sobre el contexto resaltado por ellas, suprimiendo la información de la tarea anterior [31]. Las amígdalas perciben el miedo y son capaces de alertarnos sin intervenir la consciencia para minimizar el tiempo de respuesta hacia la amenaza [33]. Las amígdalas del cerebro se encargan de producir reacciones corporales de defensa (encoger, variación del ritmo cardíaco y del flujo sanguíneo, encoger el estómago, actividad en las glándulas sudoríparas, etc.) al detectar un estímulo que consideramos que es una amenaza. Las amígdalas perciben continuamente los estímulos sean internos o externos sin que la consciencia de la persona intervenga [31]. Las amígdalas desempeñan un papel muy importante en la producción del miedo y la ira. La gran parte de las neuronas de su circuito reaccionan con los estímulos negativos [32].

- **Córtex prefrontal**: inhibe y controla las emociones [33].
- **Núcleo Accumbens**: es el centro del placer del cerebro [33].
- **Hipocampo**: permite asociar el miedo con un contexto [33], selecciona los hechos (contextos) que son importante para almacenarlo en la memoria implícita [31]. Procesa el contexto (memoria sobre los elementos que crea la situación emocional en el lugar) y envía los resultados a las amígdalas. Esto permite potenciar el control de la reacción al miedo diferenciando los distintos contextos en los que se encuentra la persona. Sin embargo, el mecanismo de detección de amenazas que subyace en nuestra consciencia aparece, aunque seamos conscientes de que el contexto en el que nos encontramos no hay ningún peligro. Esto lo experimentó Darwin en un zoológico, cuando una serpiente que estaba detrás de unos cristales realizó unos movimientos para intentar atacarle. En este momento, el cuerpo de Darwin reaccionó con miedo. Aunque él era consciente de que estaba seguro detrás de los cristales, no pudo evitar la reacción de miedo porque es un mecanismo de reacción automático que tiene nuestro cerebro [31].

2.10 IMPLEMENTACIÓN DE EMOCIONES EN UN HUMANO VIRTUAL

El artículo de Becker-Asano y Wachsmuth publicado en 2010 [35] es un trabajo que plasma los conocimientos sobre emociones desde distintas disciplinas y propone una arquitectura para crear agentes que expresan emociones primarias y secundarias. Este trabajo está centrado sobre todo en la manifestación de las emociones más intelectuales que son las secundarias. Las fórmulas definidas en este artículo han servido como base muy importante de todo el trabajo realizado en el capítulo 6.

2.11 CONCLUSIONES

En el presente proyecto, se desea plantear un mecanismo para simular el ciclo de generación de las emociones básicas en el cuerpo humano asumiendo la hipótesis de que la mayoría de las emociones básicas son fruto del proceso filogenético de la especie humana para favorecer la supervivencia de la especie. Asimismo, se desea implementar un mundo virtual que permita evaluar visualmente los efectos del mecanismo planteado. De igual modo, el trabajo se centra en la simulación de la parte no consciente del cuerpo humano y no en el proceso cognitivo desarrollado en la consciencia humana. Sin embargo, el aislamiento total de la parte consciente y la no consciente, además de ser una labor compleja, carece de realismo en la simulación de los agentes, por lo que se ha simulado la mínima parte esencial del proceso cognitivo del cuerpo y el cerebro humano, que nos permita observar el proceso de generación de las emociones básicas.

CAPÍTULO 3: **Gestión del proyecto**

Para la elaboración de este proyecto, se han utilizado los siguientes medios y políticas para cada tipo de tarea específica:

- Un ordenador portátil, para poder programar el simulador fuera de casa.
- Un ordenador de sobremesa con 2 monitores para el modelado y animación de las mallas 3D, y este ordenador también tiene instalada la aplicación Microsoft® Word para dar formato a este documento.
- Una pizarra tipo cuaderno y rotuladores de colores para dibujar diagramas conceptuales para ordenar las ideas y escribir algunas fórmulas matemáticas.
- Al ser posible, utilizar herramientas y librerías gratuitas para reducir los gastos del proyecto. Y preferiblemente, las librerías de código abierto para poder inspeccionarlas y entender mejor sus mecanismos internos. Las herramientas gratuitas utilizadas son: GIMP para la edición de texturas, Inkscape [36] para la creación de algunas ilustraciones, StarUML [37] para la creación de diagramas, y Eclipse Luna [38] como entorno de desarrollo.
- Para versionar los artefactos creados (código fuente, imágenes, este documento, modelos 3D, etc.) se ha utilizado el sistema de control de versiones Git [39]. El repositorio Git del ordenador de sobremesa se ha utilizado como repositorio de integración para unificar todos los cambios. A su vez este repositorio central está sincronizado con una cuenta gratuita de Google Drive [40] para tener una copia de seguridad.
- Un Smartphone para anotar ideas que surgen en cualquier momento y almacenarlo en Google Keep [41] en forma de notas. Y también para escribir este documento desde cualquier sitio accediendo al documento de Google Drive. Los cambios realizados en Google Drive se sincronizarán automáticamente en el ordenador de sobremesa y el ordenador portátil con la aplicación de Google Drive para Windows por lo que siempre tendremos la última versión del documento.
- Debido a la diversidad de disciplinas que abarca este proyecto, se han aprovechado las 2 horas y media del trayecto al trabajo en transporte público para leer diversos libros de temáticas dispares como gráficos por ordenador, programación, herramientas de diseño gráfico, inteligencia artificial, física, psicología cognitiva, neurología, neurobiología, mentalismo, expresiones corporales, etc. De muchos de ellos se han podido extraer ideas que se han podido aprovechar en este trabajo. Sin embargo, ha habido varios libros que

no se ha podido aprovechar, aunque eran interesantes, por salirse del ámbito que se ha definido en el comienzo de este trabajo.

Estas son las fases seguidas durante el desarrollo del proyecto:

- **a)** Búsqueda de información y adquisición de conocimientos básicos sobre tecnologías 3D. (128 horas)
- **b)** Búsqueda de librerías gráficas y opiniones en internet, y su selección. (32 horas)
- **c)** Modelado de avatar en 3D. (128 horas)
- **d)** Diseño e Implementación del simulador de entornos 3D. (1.344 horas + 12 horas de documentación de código)
- **e)** Implementación de planificador de bajo nivel en Clojure. (64 horas)
- **f)** Diseño e implementación del mundo de cajas utilizando el simulador creado. (210 horas + 16 horas de depuración)
- **g)** Modelado 3D del escenario del mundo de cajas (3 horas).
- **h)** Parametrización y puesta a punto de los agentes. (15 horas)
- **i)** Preparación y captura de resultados (20 horas) (incluye la modificación del motor para volcar resultados en fichero o pantalla)
- **j)** Elaboración de resultados y conclusiones (20 horas).

Tareas transversales: lectura de libros, artículos y documentaciones (3.600 horas). Y en cada fase se han ido anotando los detalles de cada trabajo en este documento para no perder ningún detalle (78 horas + generación y tratamiento de materiales gráficos 80 horas).

Si calculamos el coste total del proyecto:

1. Ordenador de sobremesa: 1.100 euros
2. Ordenador portátil: 680 euros
3. Smartphone: 220 euros + 56 euros por 12 meses de tarifa de datos
4. Pizarra con rotuladores: 58 euros
5. Libros: 1.800 euros
6. Personal: 1 x Ingeniero Informático x 5.750 horas x 10 euros/hora de salario (con jornadas de 8 horas de lunes a viernes excluyendo festivos) = 57.500 euros
7. 5 % de margen para imprevistos = 61.414 euros + 5 % = 64.484,70 euros
8. 5 % de beneficio = 64.484,70 euros + 5 % = 67.708,94 euros

9. 21% IVA = 67.708,94 euros + 21% = 81.927,82 euros

Obtenemos un coste total de **81.927,82 euros**.

CAPÍTULO 4: **Objetivos del PFC**

Estos son los objetivos que se desean alcanzar en este proyecto:

1. Entender las fases de creación de un entorno virtual en 3D. Diseñar e implementar un simulador que permita abordar el siguiente objetivo (punto 2). Y detallar todos los elementos necesarios para crear un entorno virtual, ya que en la mayoría de los trabajos en los que se utiliza un modelo 3D se omiten estos detalles y en realidad hay mucho trabajo por detrás. Se ha hecho esto con la esperanza de que, si mis compañeros de la profesión deciden realizar algún trabajo sobre virtualización en 3D, sin ningún conocimiento sobre el tema, les sirva de ayuda para tener una visión global de la materia, ya que para mí no ha sido fácil encontrar información orientada a principiantes sobre este campo.
2. Aglutinar los conocimientos actuales sobre las emociones desde distintas disciplinas científicas y pensar un mecanismo que permita generar estas emociones dentro de un ordenador.
3. Implementar un mecanismo que permita generar una serie de emociones en unos agentes que interactúan dentro de un entorno virtual. En concreto, se desea simular la generación de **emociones básicas** que es un conjunto pequeño de todas las emociones que conocemos, centrado en la simulación de la parte **no consciente** del cuerpo humano, y sus comportamientos más primitivos.

CAPÍTULO 5: **Diseño e implementación de un simulador de entornos en 3D**

Tras haber diseñado el avatar y el escenario siguiendo los pasos descritos en la sección 2.8, pasamos a programar los elementos de interacción en el escenario. Los lenguajes más utilizados para renderizar objetos tridimensionales son C, C++, o en algunas comunidades concretas, el HSP. Sin embargo, cada vez hay más programadores que utilizan Java [42] para este fin, por las mejoras que se han hecho en su máquina virtual y la popularidad que está teniendo en distintos ámbitos. Existen comparativas que afirman que los programas creados en Java (y ejecutadas con las últimas versiones de su máquina virtual) son casi tan rápidos como los creados en C++ [43].

Las API más empleadas y conocidas para dibujar imágenes 3D en tiempo real son Microsoft DirectX y OpenGL. DirectX ha sido durante los últimos años la tecnología más empleada para crear videojuegos para ordenadores con el sistema operativo Windows. En cambio, OpenGL tiene más tiempo de vida que DirectX y es utilizado en diversos ámbitos: investigación, simulaciones, videojuegos, películas de animación, etc.

Para la implementación del simulador 3D se ha escogido como entorno de programación Java porque es un entorno (máquina virtual incluida) con el que más experiencia tenemos. Para el desarrollo del simulador se han utilizado las siguientes librerías:

- una API gráfica llamada **jPCT** [44]
- y un motor físico llamado **JBullet** [45]

La siguiente ilustración (Ilustración 1) muestra uno de los agentes caminando dentro del entorno virtual creado:

Ilustración 1: nuestro agente puesto en movimiento dentro del simulador

5.1 JPCT

Es una librería gráfica desarrollada en Java. Permite renderizar imágenes mediante OpenGL tanto por software como por hardware. Permite la animación por esqueleto y por *keyframes*. Tiene implementado unos mecanismos de detección de colisiones sencillos. Posee cargadores para cargar ficheros 3ds, md2, obj, etc. Funciona sobre LWJGL [46]: otra librería gráfica creada para desarrollar juegos 3D en Java. La API de jPCT es una versión simplificada de LWJGL, por lo que se ha considerado que para un principiante como nosotros que empezamos a trabajar con entornos 3D, es la librería adecuada.

5.2 JBULLET

Es un motor físico implementado en Java. Es una versión portada del motor físico Bullet, que es utilizado en diversos juegos de Play Station y Xbox. Aunque un motor físico suene a algo complejo, el uso de JBullet es bastante simple. Sólo es necesario registrar los cuerpos rígidos (o blandos) en el mundo físicos con sus parámetros iniciales (posición inicial, masa, velocidad inicial, velocidad angular, la fuerza de gravedad a aplicar, etc.). Una vez inicializados y registrados todos los cuerpos, sólo hace falta avanzar el eje temporal del mundo físico. Y JBullet se encarga de calcular las fuerzas aplicadas, las nuevas posiciones, las nuevas velocidades, y las nuevas orientaciones de los cuerpos.

5.3 SISTEMA DE COORDENADAS

Antes de definir la arquitectura del simulador definiremos aquí el sistema de coordenadas que emplearemos. Cada librería gráfica y física emplean sistemas de coordenadas cartesianas

de distinto tipo, según el gusto de sus creadores. Existen dos clases de sistemas de coordenadas cartesianas 3D: dextrógiros (right-handed coordinate system) y levógiros (left-handed coordinate system) [47].

En los sistemas dextrógiros (Ilustración 2) la dirección positiva de cada eje de coordenadas se puede representar con la mano derecha. El dedo pulgar corresponde al eje X, el dedo índice al eje Y, y el dedo corazón al eje Z. Para conocer la dirección positiva de giro sobre cada eje, también se usa la mano derecha: agarrando el eje correspondiente, la dirección positiva será aquella en la que apunte el dedo pulgar. La dirección de giro es la que apunta el resto de los dedos (Ilustración 3).

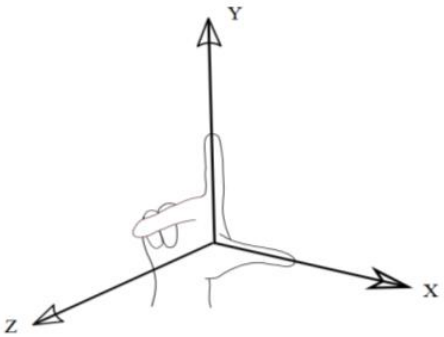

Ilustración 2: sistema dextrógiro

Ilustración 3: dirección de giro

En cuanto a los sistemas levógiros, se aplica la misma regla que en los sistemas dextrógiros, pero con la mano izquierda.

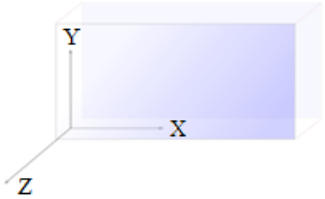

Ilustración 4: sistema de coordenadas del simulador

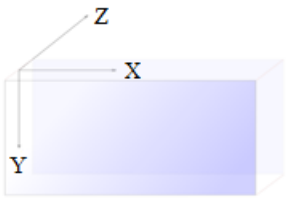

Ilustración 5: sistema de coordenadas de jPCT

Volviendo al caso de nuestro simulador; se ha utilizado el sistema dextrógiro tal y como se muestra en la Ilustración 4 (la dirección positiva del eje Z apunta hacia fuera de la pantalla). Este es el sistema de coordenadas utilizado en OpenGL [48] y también en JBullet (Bullet) [49]. Sin embargo, jPCT no utiliza el mismo sistema, aunque es parecido al sistema dextrógiro, está girado 180° sobre el eje X [50]. El eje positivo Y apunta hacia abajo y el eje positivo Z apunta hacia dentro de la pantalla (Ilustración 5). Por ello, necesitamos realizar una conversión de los puntos en el espacio cuando intercambiamos información entre el simulador y jPCT. Esta transformación es sencilla y queda definida de la siguiente manera:

$$P_{Simulador}(X, Y, Z) \leftrightarrow P_{jPCT}(X, -Y, -Z)$$
$$P_{jPCT}(X, Y, Z) \leftrightarrow P_{Simulador}(X, -Y, -Z)$$

*$P_{Simulador}(X, Y, Z)$: Un punto X, Y, Z en el espacio de coordenadas del simulador.

*$P_{jPCT}(X, Y, Z)$: Un punto X, Y, Z en el espacio de coordenadas de jPCT.

La coordenada X de un punto cualquiera en el espacio del simulador se mantiene igual al pasar al espacio de coordenadas de jPCT, y viceversa. Sin embargo, las coordenadas Y y Z deben ser multiplicadas por -1 al pasar de un espacio de coordenadas a otro, ya que los respectivos ejes apuntan en direcciones opuestas entre un espacio y el otro.

5.4 ARQUITECTURA

En el siguiente diagrama se muestra la arquitectura del simulador (Ilustración 6).

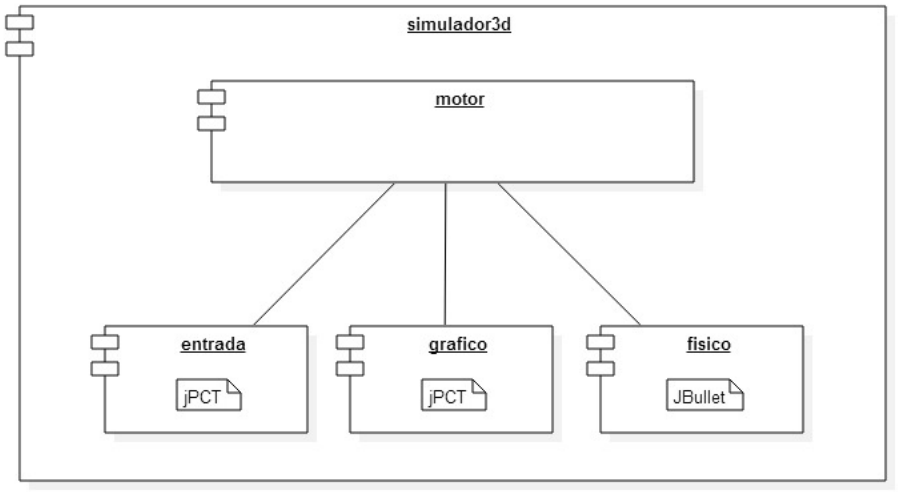

Ilustración 6: arquitectura del simulador

El diagrama muestra el núcleo del simulador 3D y todos sus subcomponentes cuyos cometidos son los siguientes:

- **motor:** es el componente encargado de configurar y ejecutar el simulador. Se implementará un motor de renderizado sencillo, para ajustar en todo momento la ratio de refresco de las imágenes para que la simulación se vea fluida y coherente respecto al tiempo de la simulación.
- **entrada**: conecta los dispositivos de entrada del usuario con el motor para notificar las acciones del usuario al simulador. Para su implementación utilizaremos el componente de entrada de jPCT.
- **grafico**: es el motor gráfico encargado de renderizar y pintar en pantalla el entorno simulado. Para su implementación utilizaremos las API gráficas de jPCT.
- **fisico**: es el motor físico encargado de aplicar las leyes físicas a los modelos del entorno para simular las colisiones entre cuerpos y sus movimientos. Para su implementación utilizaremos JBullet.

La separación entre el componente gráfico y el componente físico facilitará el cambio de las librerías subyacentes. Si en el futuro se quiere utilizar otra API gráfica u otro motor físico sólo habría que implementar el componente correspondiente, dejando intacto el resto de los componentes.

5.5 PASOS A SEGUIR PARA SIMULAR UN ENTORNO EN 3D

El componente “motor” llevará implementado todo un conjunto de API que permita efectuar los siguientes pasos necesarios para la ejecución de un simulador 3D (pasos A al H extraídos de [13]):

A. Inicialización del simulador
B. Carga de los modelos
C. Configurar la cámara
D. Configurar la proyección
E. Configurar los focos de luz
F. Identificación de los objetos en el campo de visión
G. Renderizado
H. Ajuste del *frame rate*
I. Ejecución del bucle principal de simulación

A. Inicialización del simulador

Creamos la configuración de nuestro simulador y se lo pasamos como parámetro al simulador y lo ejecutamos. Esto configurará la tarjeta gráfica y mostrará los gráficos 3D en pantalla. A partir de este momento se podrá utilizar su API, antes no funcionaría.

B. Carga de los modelos

Cargamos los modelos desde los ficheros externos utilizando los cargadores que nos proporciona la API. Cada librería soporta unos formatos determinados. Los más típicos son .3ds, .obj o .md2.

Una vez cargados los modelos, en algunas API hace falta cargar las texturas por separado, si es así, lo cargamos también en este paso. Los formatos más típicos de las texturas son .tga o .png con canal de transparencia (alpha).

Lo siguiente es colocar cada modelo en su punto de partida. Primero el escenario, y luego los personajes y otros objetos del entorno.

C. Configurar la cámara

Creamos la cámara que nos permite mostrar el mundo virtual a través de la pantalla. Una vez creada hay que situarla en su punto inicial e indicar la dirección hacia donde enfoca esta.

D. Configurar la proyección

En este punto se define el campo de visión de la cámara y la distancia mínima y máxima en las que la cámara será capaz de mostrar los objetos. En el campo de visión, se suele poner un ángulo de 45° o 60° [13].

Puesto que no podemos ver los objetos que están muy cerca (suele aparecer desenfocado y borroso) ni muy lejos, indicamos una distancia mínima y otra máxima. El plano que se encuentra en la distancia mínima desde la cámara se conoce como *near clip* y todos los objetos que estén por delante de él, no aparecerá en la escena. Por el contrario, el plano que se encuentre a la distancia máxima se conoce como *far clip*, donde todo aquel objeto situado por detrás de él tampoco se podrá visualizar en la escena [17]. Estos valores se configuran en función de la naturaleza del escenario que se quiera mostrar. Si la acción está siempre en primer plano no hace falta poner una distancia máxima muy grande. Pero si la acción transcurre a lo largo del escenario es conveniente poner una distancia máxima mayor para permitir ver las acciones que transcurren tanto en primer plano como en un lugar más lejano del escenario. El principal inconveniente de poner una distancia mayor es que aumenta la superficie en el cual debe realizar cálculos el simulador y esto consume mayor cantidad de recursos, por lo que puede llegar a ralentizar toda la escena. Si aumentamos el ángulo de visión pasaría algo similar [13].

E. Configurar los focos de luz

Los focos de luz son necesarios para poder visualizar los objetos del entorno. Sin las luces los objetos aparecen en negro. Al incluir un foco luminoso, se visualizarán los colores de la superficie de los objetos al reflejarse la luz sobre ésta. Cuando se crea una luz se le indica la dirección, con la cual se permite regular su intensidad. La luz es más luminosa cuando su dirección es perpendicular a la superficie que la refleja. Su intensidad va disminuyendo a la vez que se va inclinando la dirección de la luz. Cuando la superficie del objeto esté alineada con la dirección de la luz el objeto no será visualizado ya que la

luz no es reflejada. Sucederá lo mismo si la luz avanza en contra del vector normal de la superficie del objeto [51] [52].

F. Identificación de los objetos en el campo de visión

Al identificar sólo a los objetos que se encuentran en el campo de visión de la cámara, se está acelerando el procesamiento de la escena. Esto es debido a que los objetos que estén fuera de escena se pueden ignorar y no procesarlos (esto es sólo en el caso del modelo visual, los modelos físicos se deben procesar siempre ya que las leyes físicas se deben aplicar para todos los elementos del mundo virtual).

G. Renderizado

Es en esta fase cuando se genera una imagen en dos dimensiones a partir de nuestro modelo tridimensional para ser pintado en pantalla.

H. Ajuste del frame rate

El *frame rate* es la tasa de refresco de los fotogramas generados en el paso anterior. Debido a que en unas escenas se consume más tiempo de procesamiento que en otras, el tiempo de actualización de la imagen entre fotograma y fotograma no es constante. Por ello, hay que ajustar el *frame rate* para que esto sea constante (o casi constante) y así conseguir que los objetos se muevan fluidamente en pantalla sin saltos.

I. Ejecución del bucle principal de simulación

Tras configurar los distintos parámetros del simulador, invocamos el bucle principal de simulación. Éste se compone de las siguientes fases:

1. Primero de todo se comprueba si existe algún mensaje procedente del usuario que indique que la simulación debe finalizar. En tal caso, el simulador liberará todos los recursos que ha reservado y finalizará la simulación. En caso contrario ejecutará el siguiente paso.
2. Procesar los eventos de entrada del simulador. Estos eventos vienen de teclado, ratón u otros dispositivos de entrada.
3. Actualizar los modelos 3D del entorno utilizando el tiempo transcurrido desde la última actualización y aplicando las órdenes, si existen, recibidas por los dispositivos de entrada en la fase anterior.

4. Ya en la última fase, se renderiza el modelo actualizado. Para el renderizado es necesario ejecutar las siguientes sub-fases para poder pintar correctamente en la pantalla del ordenador:
    i. Limpiar el buffer: limpia la imagen que se renderizó en algún paso anterior para pintar una nueva.
    ii. Escribir la imagen renderizada en el buffer limpiado.
    iii. Actualizar el buffer para preparar su salida a la pantalla.
    iv. Y finalmente dibujar la imagen del buffer en la pantalla.

5.6 MOTOR DE RENDERIZADO

En los primeros juegos diseñados para ordenador, la velocidad de desplazamiento de los personajes era dependiente de la frecuencia de reloj del microprocesador. Por ejemplo, cuando lanzábamos estos juegos en un 386 o 486 y pulsamos el botón "turbo", los personajes del juego se movían con más velocidad. Este es un claro ejemplo de un motor de renderizado mal diseñado. Para minimizar las diferencias entre un ordenador y otro, el propio motor de renderizado debe tener un reloj interno que dirija el avance de la simulación. Y así desligarnos de la frecuencia de reloj del microprocesador.

En nuestro motor de renderizado (componente "motor") se tienen en cuenta dos medidas para mantener su fluidez: los fotogramas por segundo (FPS: *frames per second* [43]), y las actualizaciones por segundo (UPS: *updates per second* [43]). La FPS mide el número de fotogramas que el motor gráfico ha pintado por pantalla en cada segundo (en media). Y la UPS mide el número de veces (en media) que se ha actualizado el mundo físico de JBullet.

El motor de renderizado debe intentar absorber las diferencias de tiempo entre un ciclo y otro, manteniéndolas constante para evitar que la simulación avance más rápida cuando la CPU tenga que realizar pocos cálculos, y más lenta cuando haya que realizar más cálculos. Para ello, se mantiene contante la UPS a una frecuencia U, que es configurada justo antes de lanzar la simulación. La FPS será sacrificada, omitiendo la fase de renderizado y pintado en pantalla, cuando la carga de trabajo en un ciclo del motor sea intensa. Como resultado, UPS estará siempre próxima a U, pero la FPS puede ser igual a U cuando no haya mucha carga en el ordenador e inferior cuando la carga sea mayor. De esta forma permitimos que la simulación en el entorno físico no se atrase en el tiempo y por lo tanto evitar saltos bruscos e incoherencias en las colisiones entre modelos. Por ejemplo, si la UPS es muy baja en un entorno en el que existen cuerpos móviles que se desplazan a una velocidad muy alta, puede ocurrir que estos cuerpos atraviesen las paredes. Esto ocurre cuando la distancia desplazada,

en un sólo ciclo, sea lo suficientemente grande como para posicionarse en el otro lado de la pared cuando en el ciclo anterior estaba detrás. Al no haber una intersección entre la pared y el cuerpo móvil, el motor físico no es capaz de detectar la colisión, y esto se conoce como túnel (*tunneling*) [53].

Sin embargo, la FPS la podemos hacer variable mientras que no sea inferior a 24 (se ha tomado como referencia el número de fotogramas que se proyectan en las películas que son de 24 FPS [43]). Lo suficiente para que nuestro cerebro capte la imagen de la pantalla como una imagen en movimiento (para nuestra simulación no necesitamos más). Tampoco es necesario intentar conseguir una FPS superior a la frecuencia de refresco de nuestra pantalla ya que los fotogramas sobrantes no van a ser pintados en ella.

La UPS utilizada en nuestro motor es fija y es 30, ya que no hay movimientos rápidos y complejos no necesitaremos más. Si queremos aumentar la FPS porque estamos por debajo de 24, entonces necesitamos bajar la UPS para liberar recursos del ordenador a favor del renderizado de imágenes. Esto es una labor de puesta a punto para conseguir un equilibrio entre el FPS y el UPS, necesaria para simulaciones complejas que requieren mayor carga de procesamiento.

A continuación, se exponen los detalles de implementación. Para procurar mantener la UPS constante, se calcula primero la franja de tiempo disponible en cada iteración. Si, por ejemplo, queremos que el motor de renderizado funcione con UPS=60, la franja de tiempo disponible en cada iteración sería de unos 16,7 ms aproximadamente. Una vez obtenido esto, sabemos que en un caso ideal se cumple siempre la siguiente fórmula:

$$\boldsymbol{Taf + Trp \leq 16{,}7\ ms}$$

***Taf**: tiempo consumido en actualizar el mundo físico.*

***Trp**: tiempo consumido en renderizar y pintar por pantalla.*

Pero como ya se ha señalado antes, *Taf* y *Trp* pueden aumentar y disminuir dependiendo de lo que esté ocurriendo dentro del simulador. Y por ello, habrá ocasiones en las que la fórmula anterior no se cumpla. Cuando *Taf* + *Trp* es menor que 16,7 ms dormimos el hilo en el que se está lanzando el motor hasta completar los 16,7 ms. Si *Taf* + *Trp* es igual a 16,7 ms no haremos nada especial y pasamos a la siguiente iteración. El problema surge cuando *Taf* + *Trp* es mayor que 16,7 ms, porque en este caso, se consume más tiempo del que se dispone. Para corregir este exceso de tiempo, lo que se hace es, ir sumando el tiempo excedido dentro de una variable global e ir compensando estos excesos en las siguientes iteraciones. Para compensar el UPS disminuido, se actualiza más de una vez el mundo físico en cada iteración. Si el tiempo excedido acumulado hasta el momento supera los 16,7 ms, actualizaremos el mundo físico dos veces: una por el ciclo actual y otro por el ciclo excedido de 16,7 ms. Cuando se produce esta corrección se resta 16,7 ms de la variable global 'exceso' por cada actualización correctora que se realice dentro del mismo ciclo. Y en estos casos se omite la fase de renderizado y pintado por pantalla, para evitar exceder la franja de tiempo del ciclo actual, disminuyendo el FPS (Ver Ilustración 7). En este ejemplo el UPS es 60, por lo que el FPS tendrá como máximo 60 también, y debe perder unos 37 fotogramas por segundo para que nos encontremos con el escenario "malo" mencionado. Es decir, en más de la mitad de los ciclos debe estar corrigiendo el UPS. Para los escenarios simples con los que vamos a trabajar, la pérdida es de entre 3% y 10%, dependiendo del hardware.

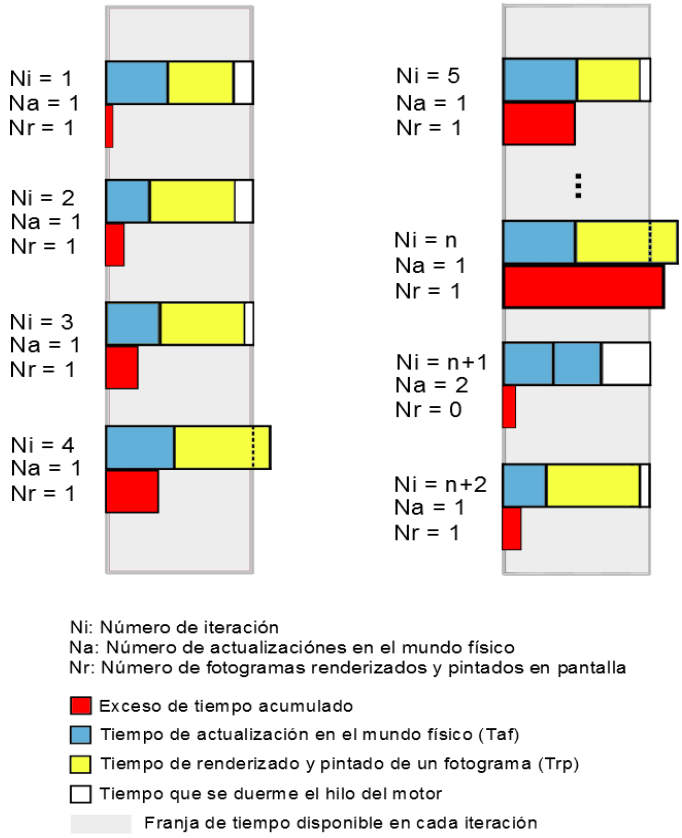

Ilustración 7: explicación gráfica sobre el mecanismo de corrección de UPS

Es importante destacar que el exceso de tiempo no se produce sólo cuando, *Taf* + *Trp* > 16,7 ms. Sino también, cuando hacemos dormir el hilo del motor. Cuando invocamos el método *sleep()* de un *Thread*, el hilo no suele dormir el tiempo exacto indicado, sino un poco

más. Este un “poco más” es el exceso que iremos sumando en la variable global de exceso, y se corregirá en los posteriores ciclos cuando sea mayor que 16,7 ms.

En JBullet la actualización del mundo físico se realiza mediante la invocación del método: *stepSimulation()* de la clase *DynamicsWorld*. Hay que tener cuidado con el método que invocamos ya que existen 3 distintos:

1. *public int stepSimulation(float timeStep)*
2. *public int stepSimulation(float timeStep, float maxSubSteps)*
3. *public int stepSimulation(float timeStep, float maxSubSteps, float fixedTimeStep)*

Para nuestro simulador ha habido que invocar al método 2 con el parámetro *maxSubSteps* a cero. Esto es necesario porque JBullet tiene, por defecto, un contador de reloj interno que realiza una actualización cada 1/60 segundos, es decir, tiene una frecuencia de actualización de 60Hz. Pero como ya hemos dicho anteriormente, nuestro simulador tiene un UPS fijo de 30, y no nos sirve. Para no utilizar el contador de reloj de JBullet, es necesario pasar el valor cero al parámetro *maxSubSteps*. Y además el primer parámetro: *timeStep* hay que mantenerlo constante porque la responsabilidad de calcular el tiempo de cada ciclo pasa de JBullet a nuestro motor de renderizado. De no hacerlo así estaríamos duplicando el cálculo y JBullet puede llegar a actualizar el mundo físico más número de veces que los esperados.

5.7 MOTOR DE SIMULACIÓN

El funcionamiento básico del motor de simulación se puede observar en el siguiente diagrama de secuencia (Ilustración 8).

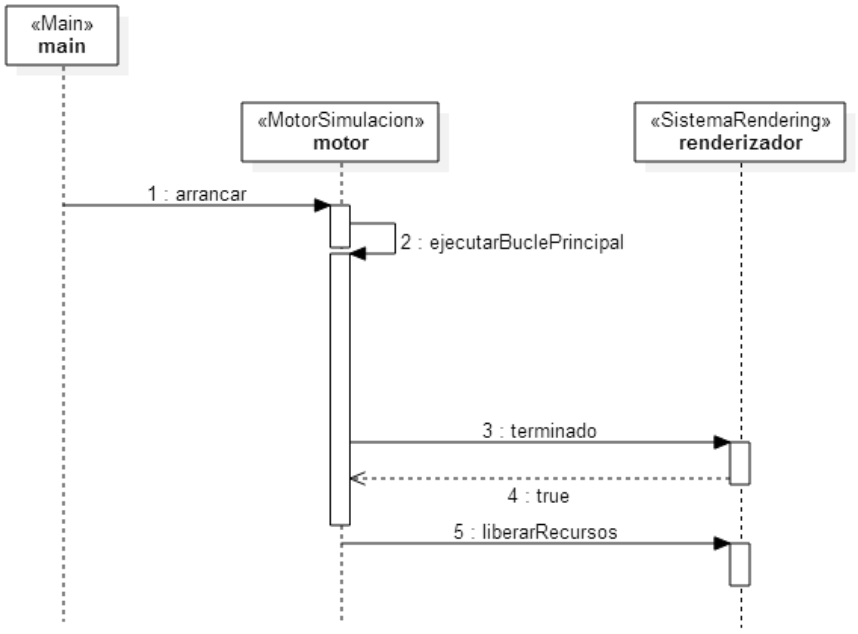

Ilustración 8: interacción entre las clases del motor de simulación

Desde la clase *Main* se instancia el *MotorSimulacion* y a continuación se arranca el simulador (1). Al arrancar se ejecuta el bucle principal de simulación (2) hasta que el usuario decida parar la simulación. Cuando el usuario envía la señal de fin de simulación al *SistemaRendering* a través del interfaz de usuario, entra en la rutina del motor que pregunta en cada iteración del bucle principal si la simulación debe ser finalizada (3). El *SistemaRendering* devuelve true en dicha consulta (4), el motor de simulación sale del bucle principal e invoca al método *liberarRecursos* (5) del *SistemaRendering* para liberar los recursos utilizados en la tarjeta gráfica, la memoria y los hilos que procesan las tareas en segundo plano, antes de cerrar la aplicación.

Ahora vamos a dirigir nuestra atención en lo que ocurre dentro del bucle principal de simulación. En el siguiente diagrama de secuencia (Ilustración 9) se puede observar la interacción entre las clases participantes y los pasos que se ejecutan dentro del bucle principal.

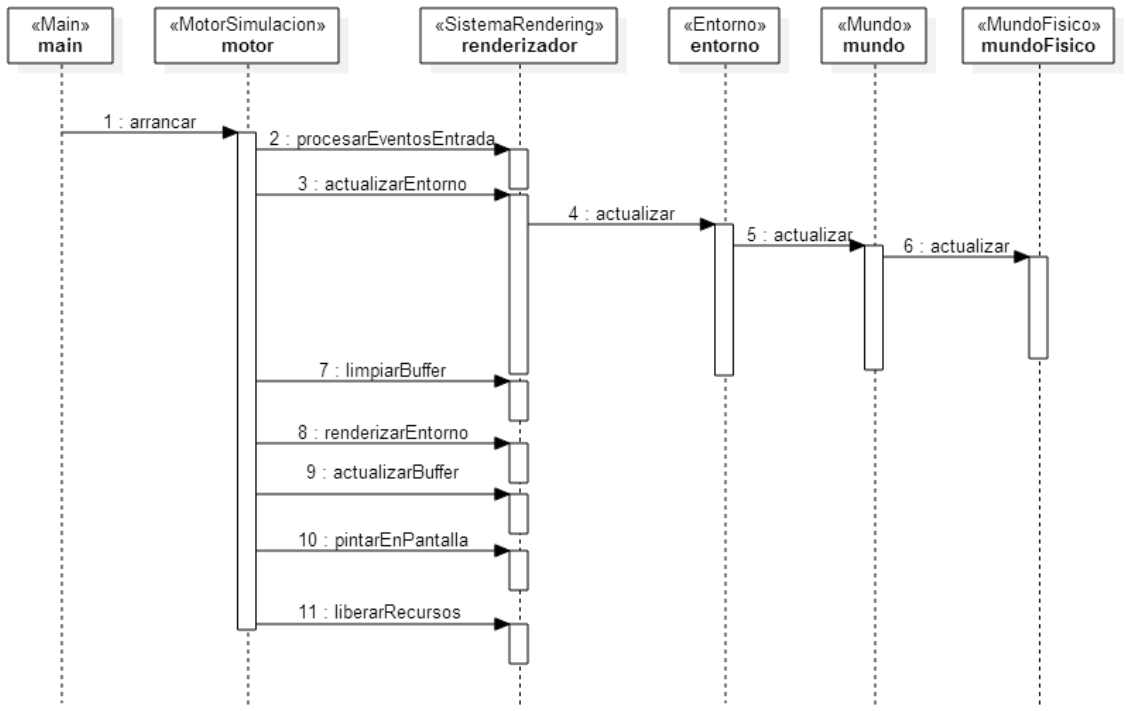

Ilustración 9: secuencia de ejecución para avanzar un ciclo en el motor físico

En cada iteración del bucle de simulación, se procesan los eventos que se han generado por el interfaz de usuario (movimientos del ratón, los desplazamientos de la rueda del ratón, y teclas presionadas en el teclado). Una vez procesados los eventos de entrada (2), indica al *SistemaRendering* que ejecute un ciclo de simulación en el *Entorno* mediante el método *actualizarEntorno* (3). En este instante, el motor de simulación se encarga de realizar las transformaciones solicitadas por el usuario en el Entorno y/o en el *SistemaRendering* a través de los eventos de entrada (Por ejemplo, mostrar la información de depuración en pantalla o

mover la cámara dentro del entorno). La actualización de entorno desencadena una secuencia de llamadas que pasa al *Entorno* (4), después al *Mundo* (5) y finalmente al *MundoFisico* (6). Dentro del *MundoFisico* se ejecuta una simulación invocando a la API de JBullet y este aplica las propiedades físicas halladas a todas las entidades físicas que están registradas. Esto hace que los cuerpos se desplacen y giren. Una vez actualizadas todas las entidades del entorno, se pasa el testigo del motor físico al motor gráfico. Primero se limpia la zona de memoria que se va a utilizar para renderizar la escena actual (7). Tras limpiar el buffer, indicamos al motor gráfico (jPCT) que recopile las entidades que hay en el *Mundo* y las renderice en el buffer (8). Cuando el buffer esté completado con la información de la escena renderizada, se indica al motor gráfico que utilice dicho buffer para pintar la escena en pantalla (9). Y finalmente, se pinta el contenido del buffer en pantalla (10). En el caso de que el usuario haya solicitado terminar la simulación, es en este punto donde se comprueba este hecho. Y si el *SistemaRendering* indica que la simulación ha terminado, se invoca al método *liberarRecursos* (11) y finalizaría la simulación.

Como se ha podido observar, en la clase *MotorSimulacion* está implementado el control del *frame rate* mencionado en el capítulo 5.6. Para ello, se apoya en la clase *SistemaRendering* que se encarga de manejar el motor gráfico (ver Ilustración 10). Es importante destacar el hecho de que las llamadas 7, 8, 9 y 10 del diagrama anterior no se ejecutarían en el caso de que haya que saltar el pintado en pantalla, porque en el ciclo anterior se haya empleado más tiempo del que se dispone en cada ciclo de ejecución. Precisamente la clase *Reloj* se encarga de medir el tiempo transcurrido entre un ciclo y el siguiente.

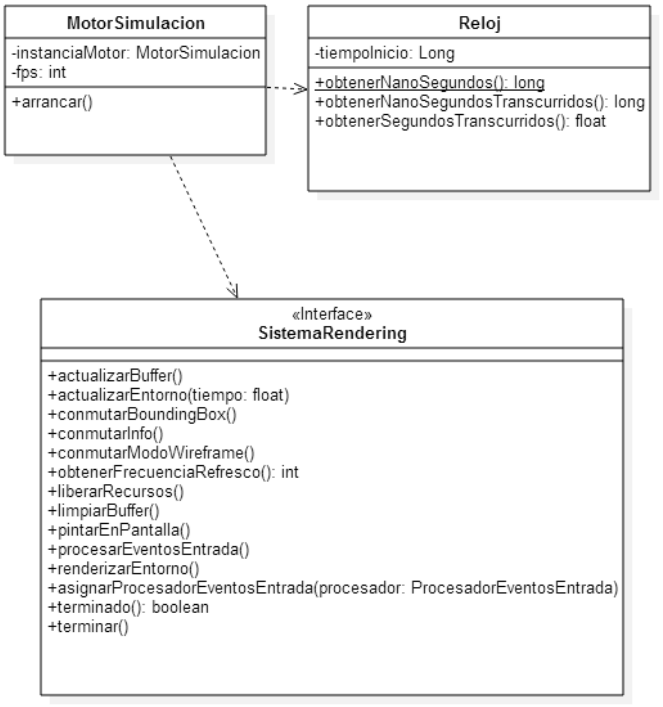

Ilustración 10: el motor de simulación encargado de mantener el *frame rate*

5.8 EVENTOS DE USUARIO

Una vez descrito el ciclo básico de ejecución. A continuación, se describe con más detalle la interacción de las interfaces de usuario con el *SistemaRendering*. Las clases implementadas permiten recoger los eventos de entrada del usuario y se pueden observar en el siguiente diagrama de clases (Ilustración 11).

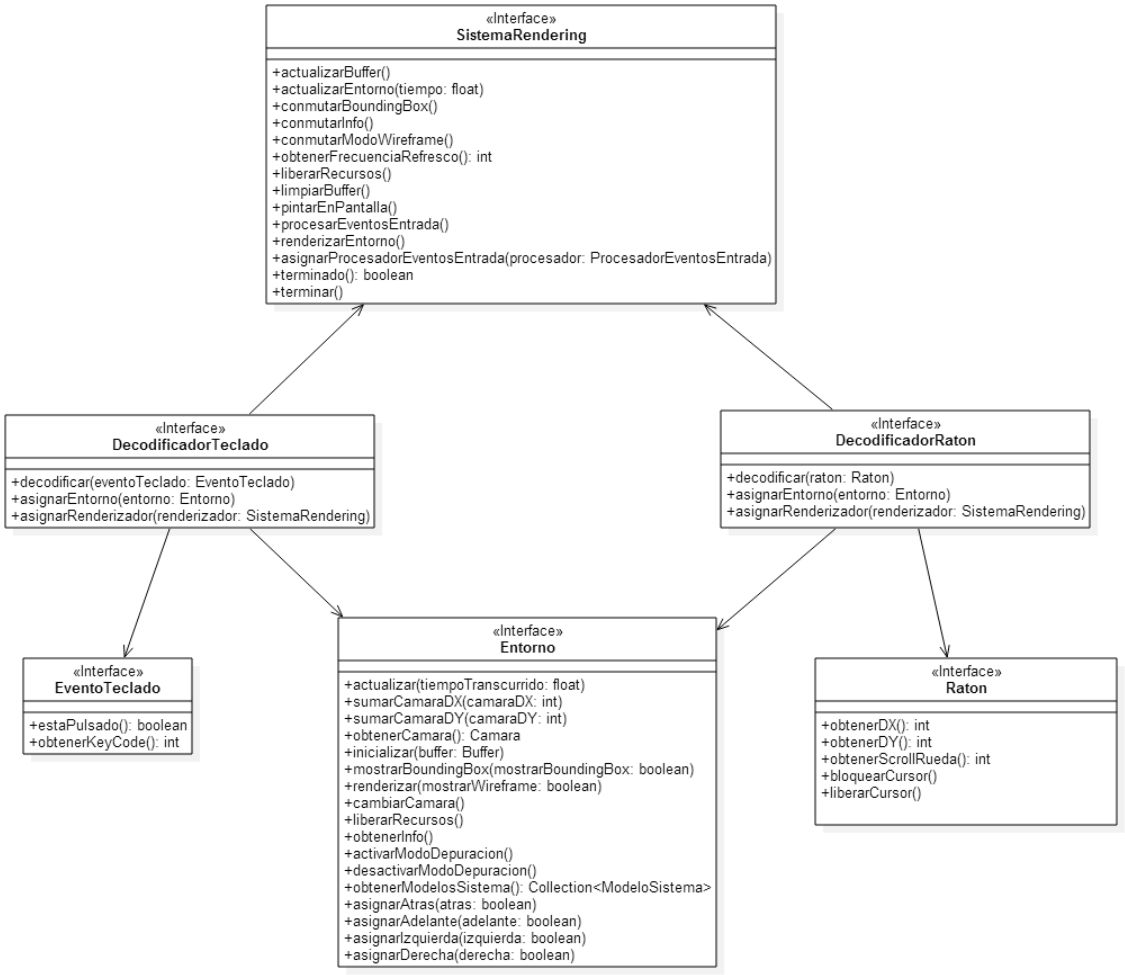

Ilustración 11: relación entre clases que permiten escuchar las interfaces de usuario

Como se puede observar en el diagrama, se ha dado soporte a los dispositivos de entrada más básicos como es el teclado y el ratón. Las interfaces *DecodificadorTeclado* y *DecodificadorRaton* se encargan de procesar los eventos de teclado y ratón respectivamente, transformándolos en instrucciones que se ejecutan en el *Entorno* y/o *SistemaRendering*. Los eventos de teclado son encolados y se hace *polling* en cada ciclo de ejecución (este mecanismo viene ya implementado en jPCT). En cambio, los eventos de ratón no se encolan, sino que se coge el estado del ratón en el momento de su decodificación. En el caso del desplazamiento de la rueda del ratón, se obtiene el contador de cuánto se ha desplazado hacia arriba o abajo obteniendo un valor positivo, si el desplazamiento es hacia arriba, y negativo si el desplazamiento es hacia abajo.

5.9 SINCRONIZACIÓN ENTRE MOTORES FÍSICO Y GRÁFICO

Hasta ahora hemos visto que el *SistemaRendering* actúa indirectamente, a través de la clase *Entorno*, sobre el motor físico (JBullet) en cada ciclo de ejecución del bucle principal del simulador. Pues bien, a continuación, se detalla cómo se reflejan las transformaciones (desplazamiento y rotación) aplicadas en las entidades físicas, sobre las entidades visuales. Para ese fin, se han representado las entidades que participan en esta fase en el diagrama de clases de la Ilustración 12.

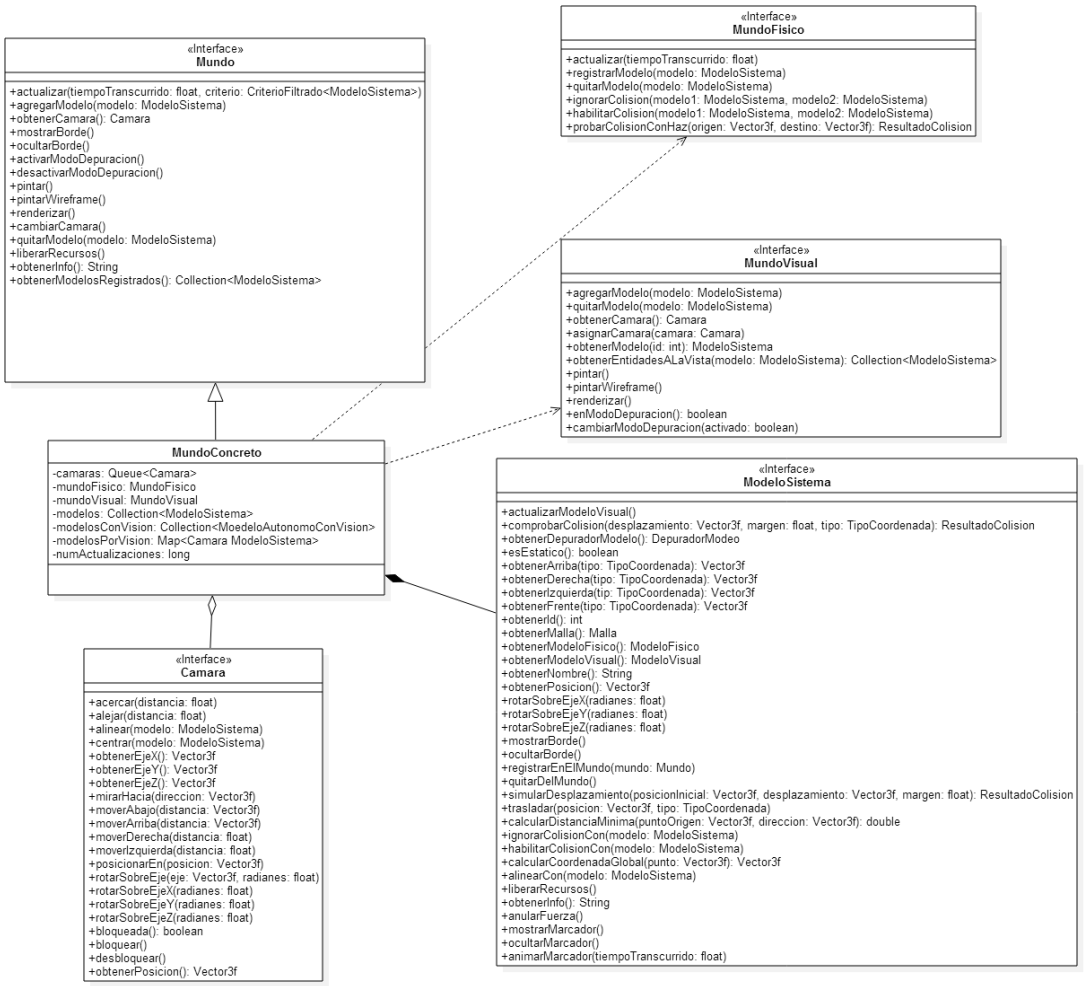

Ilustración 12: relación entre los mundos visual y físico

El *Mundo* está formado por dos partes: el *MundoFisico*, y el *MundoVisual*. Esto se debe a la decisión de arquitectura tomada anteriormente, y facilita el cambio de motor tanto gráfico como físico, al discernir claramente la frontera entre un motor y el otro. El *Mundo* se encarga de distribuir las tareas entre el motor físico y el motor gráfico dependiendo de la operación que se quiera realizar sobre ella.

Dentro del *Mundo*, las entidades se representan como instancias de *ModeloSistema*. Cada *ModeloSistema*, a su vez, tiene una parte visual y física de forma análoga al *Mundo* y por las mismas razones.

Una vez visto el diagrama de clases, pasamos a ver la interacción detallada en el siguiente diagrama de secuencia (Ilustración 13).

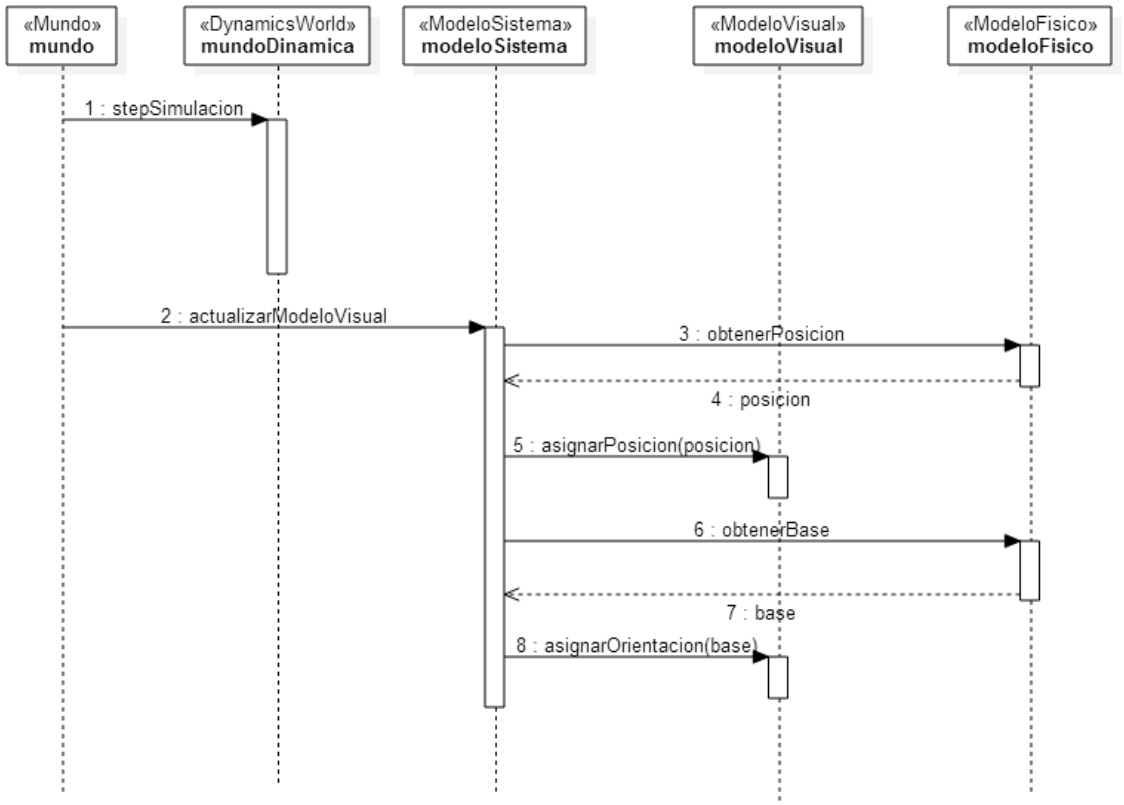

Ilustración 13: secuencia de ejecución de un paso en el Mundo del simulador

Cuando la entidad *Entorno* invoca el método *actualizar()* sobre el *Mundo*, éste hace una llamada al método *stepSimulation(...)* en el mundo físico del JBullet llamado *DymanicsWorld* (1). Al terminar la simulación en el mundo físico, el *Mundo* del simulador invoca para cada *ModeloSistema* registrado en él, el método *actualizarModeloVisual* (2). Al realizar esta llamada, cada *ModeloSistema* obtiene del *ModeloFisico* la nueva posición del modelo (3) y lo asigna al *ModeloVisual* asociado (5). Y seguidamente, obtiene la nueva orientación del *ModeloFisico* (6) y la asigna a su *ModeloVisual* (8). Así cuando el *SistemaRendering* vaya a dibujar en pantalla la nueva escena, todos los modelos visuales están sincronizados con el estado del *MundoFisico*.

5.10 DETECCIÓN DE COLISIONES

En esta sección se aclarará, brevemente, lo que ocurre dentro de JBullet cuando se invoca al método *stepSimulation(…)* en la clase *DymanicsWorld* que acabamos de ver en el apartado anterior.

Como se ha mencionado anteriormente, nuestro simulador tiene separado el modelo visual, del modelo físico. Una de las razones por las que se ha tomado esta decisión es para mejorar el rendimiento del simulador simplificando el problema. Para el modelo visual, las mallas de polígonos utilizadas suelen ser muy complejas, con zonas convexas y cóncavas. Y realizar pruebas de colisión sobre todos estos polígonos son muy costosas. Esto sería adecuado para simulaciones en las que requiera una alta precisión y que la resolución del cálculo no tenga que ser inmediata. Sin embargo, en nuestro caso no necesitamos tanta precisión, sino resultados inmediatos para reflejarlos en pantalla en tiempo real. Para simplificar esto, el modelo físico contiene una malla más sencilla que recubre el modelo visual, conocido como "volumen delimitador" (*bounding volume*), con mucho menos número de polígonos y de geometrías simples (si el modelo visual tiene una geometría simple como, un cubo, una esfera o un disco, el número de polígonos será igual en ambos modelos). Así, el número de comprobaciones que hay que realizar en cada modelo disminuye considerablemente, no sólo por el número de polígonos que ha disminuido, sino también por el tipo de operaciones aritméticas que se realizan para hallar las intersecciones entre estos modelos.

Sin embargo, en el mundo físico suele haber varios modelos de distintos tamaños y formas, y pueden colisionar todos en cualquier momento. Por lo que, en cada iteración del motor físico, necesitaremos realizar $(n-1) + (n-2) + \cdots + 1 = n(n-1) / 2$ comprobaciones, siendo *n* el número de modelos físicos que hay en el entorno. Esto tiene una complejidad de $O(n^2)$ y no es viable su uso para nuestro simulador ya que el tiempo de cálculo crece de forma cuadrática para cada modelo físico añadido. Para disminuir este tiempo de cálculo, en los simuladores de tiempo real (por ejemplo, en los videojuegos), se divide la detección de colisiones en dos fases: *fase amplia* (*broad-phase*) y *fase estrecha* (*narrow-phase*) [53]. En la *fase amplia,* se identifican los grupos de polígonos que pueden estar colisionando, y los agrupa en bloques asegurando que estos bloques sean disjuntos con el resto (ver Ilustración 14). Y en la *fase estrecha*, se calculan las intersecciones exactas entre los modelos que están dentro de estos bloques. La *fase estrecha* sigue requiriendo una comprobación por pares de modelos (de complejidad $O(n^2)$). Sin embargo, al estar agrupados en pequeños bloques de modelos físicos, n suele ser bajo, y por lo tanto se reduce bastante el tiempo de cálculo empleado en cada iteración. Existen varios métodos para crear estos grupos de modelos en

la *fase estrecha*. Por ejemplo, en la bibliografía [53], se muestran técnicas para dividir el espacio mediante rejillas o árboles. Para más detalle se recomienda consultar la bibliografía mencionada.

Ahora vamos a describir los *volúmenes delimitadores*. Estos suelen ser polígonos simples: esferas o cubos. Pero también polígonos convexos más complejos, perjudicando, claro, el rendimiento del simulador porque incrementa el número de operaciones que hay que realizar para hallar las colisiones. Las características más deseables que debería tener un *volumen delimitador* según [53] son:

- Bajo coste de cálculo para hallar las intersecciones
- Mejor ajuste a la malla del modelo visual
- Facilidad para rotar y transformar
- Bajo consumo de memoria

Teniendo en cuenta estas características describimos a continuación, los volúmenes delimitadores más utilizados en los simuladores en tiempo real.

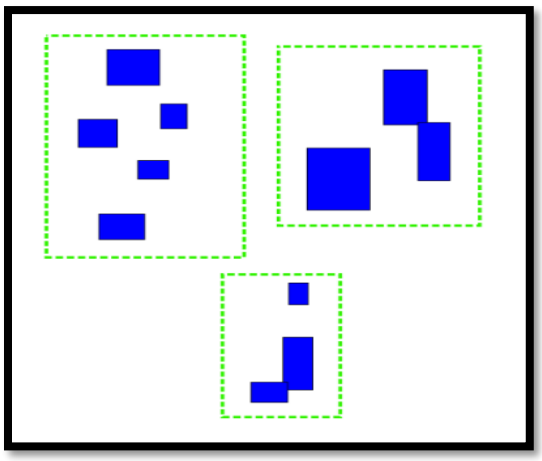

Ilustración 14: agrupación de modelos en la *fase amplia*

La esfera es el volumen que permite detectar las colisiones más rápidamente. Una esfera consume sólo cuatro números decimales en memoria: tres para almacenar el centro y uno para almacenar el radio. Para calcular la intersección con otra malla, hay que calcular la distancia entre el centro de la esfera y la otra malla. Comparando la distancia obtenida con el radio de la esfera, es fácil saber si las mallas están solapando o no. La principal desventaja de esta geometría es la imprecisión de las colisiones cuando se recubre una malla visual no esférica (ver Ilustración 15). Por ejemplo, si recubrimos una malla visual que tiene forma de pirámide con una esfera en el modelo físico (ver Ilustración 16), se producirán colisiones

falsas (falsos positivos) en la mayoría de las ocasiones por la gran diferencia que existe entre la forma de la malla visual y el modelo físico.

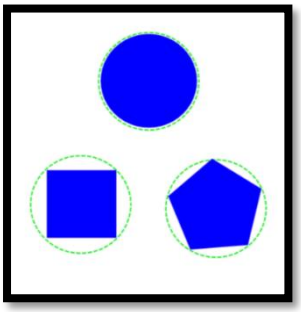

Ilustración 15: distintos tipos de polígonos recubiertos por una esfera

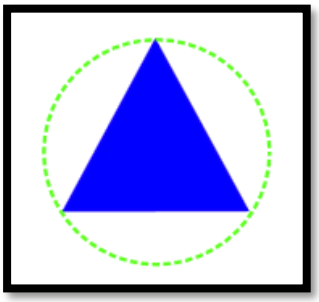

Ilustración 16: una pirámide recubierta por una esfera

La segunda geometría eficiente es la AABB (*axis-aligned bounding box*). Una AABB es un hexaedro cuyos vectores normales de cada cara son paralelos con los ejes de coordenadas del mundo virtual que se denomina ***coordenada global***. Estos hexaedros consumen el espacio en memoria de seis números decimales. Pero se pueden representar de tres formas distintas:

1. Por dos puntos: son las coordenadas máximas y mínimas de cada eje.
2. Por un punto y tres longitudes: la coordenada de la esquina con menor distancia con respecto al origen, y las longitudes hacia las tres esquinas vecinas.
3. Por un punto y tres longitudes medias: el punto central del hexaedro, y las distancias desde éste hasta las caras para cada uno de los tres ejes X, Y Z.

El uso de las AABB facilita el cálculo de intersecciones cuando el otro modelo físico también es una AABB. En este caso es inmediato, teniendo las coordenadas mínimas y máximas de cada AABB. Sólo es necesario saber si en alguno de los ejes no se produce el solapamiento. Esto es una condición suficiente para asegurar que no se produce ningún solapamiento entre ambos. Dicho de otra forma: si se detectan solapamientos **en los tres ejes a la vez**, significa que los dos AABB están colisionando.

La tercera geometría es similar a la segunda, pero menos eficiente. Se trata de las OBB (*oriented bounding box*) y son hexaedros alineados a los ejes de coordenadas, pero a diferencia de las AABB, estas se alinean con los ejes de coordenadas de la malla, conocida como **coordenada local** (Ilustración 17). Cada modelo físico se representa dentro de su propio sistema de coordenadas. Sin embargo, desde el punto de vista de un observador que esté manipulando todos estos modelos, éste maneja la información en su propio sistema de coordenadas, distinto al de cada malla, por lo que siempre es necesario hacer una conversión para pasar estos modelos al sistema de coordenadas del observador. Antes de procesar los modelos, tanto si es para renderizarlos como para comprobar sus intersecciones, las coordenadas locales de cada modelo son transformadas y se pasan al sistema de coordenadas del mundo (visual o físico, dependiendo de si estamos renderizando o comprobando las intersecciones) para utilizar la misma referencia. Esta referencia es la que hemos llamado ***coordenada global*** en el apartado de las AABB. Un OBB se representa mediante quince números decimales: tres para representar el punto central del hexaedro, nueve para representar los tres ejes en su coordenada local, y tres para indicar las longitudes en cada eje desde el punto central. Como se puede notar, el consumo de memoria ha aumentado considerablemente. En cuanto a las pruebas de colisión, aumenta considerablemente la complejidad de cálculo. El detalle lo vamos a omitir porque se sale del alcance de este documento. Pero el algoritmo detallado se puede consultar en el apartado "*4.4 Oriented Bounding Boxes (OBBs)*" del libro [53].

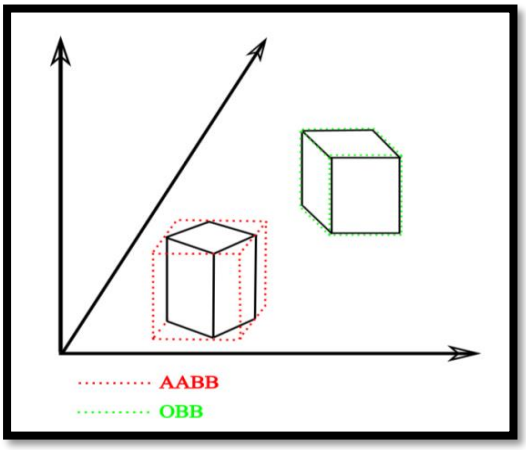

Ilustración 17: diferencias entre AABB y OBB

Y, por último, para mallas de polígonos mucho más complejas se puede utilizar una envolvente convexa (*convex hull*). Mediante el uso de una envolvente convexa podemos representar todo tipo de mallas que sean convexas. Sin duda esta es la más ajustada a la malla del modelo visual y por lo tanto el modelo físico más preciso con respecto a los anteriores.

Sin embargo, es la más costosa computacionalmente a la hora de realizar comprobaciones de colisión. Para conocer los algoritmos más utilizados para calcular las intersecciones para este tipo de mallas, podemos echar un vistazo al capítulo 9 del libro [53].

De cara a nuestra aplicación, JBullet soporta los cuatro tipos de volúmenes delimitadores mencionados. Y los correspondientes algoritmos para detectar las colisiones están ya implementados, por lo que nos ahorramos una gran cantidad de tiempo y esfuerzo.

CAPÍTULO 6: **Mundo de cajas**

En la aplicación a implementar se van a ejecutar varios agentes idénticos, pero con distintos parámetros. En el comienzo de la simulación, aparecerá una caja colocada en un lugar del escenario. Cada agente recibirá una misión en la que se indica un punto en el que éstos deberán colocar la caja. Si el agente completa la misión se quedará dando vueltas por el escenario para comprobar que la caja sigue estando donde la colocó, ya que los otros agentes pueden moverla del sitio. La simulación no tiene un final definido, por ello, sólo se podrá terminar si cerramos la ventana del simulador. Los objetivos que se quieren alcanzar con la implementación de esta aplicación son:

1. Hacer visible e interpretable, por nuestra vista, la emoción del agente mediante expresiones corporales.
2. Ver cómo su emoción va cambiando al interactuar con el entorno.

Cada agente es capaz de percibir los estímulos externos a través de la vista. Y por simplicidad, el agente es capaz de identificar un modelo visto sin necesidad de evaluar sus formas; en la mente del agente se registra cada modelo visto por su nombre y por ello es capaz de distinguir entre un agente y otro, aunque sus apariencias sean las mismas. El entorno en el que se desenvuelve el agente es inaccesible [54] porque la información que tiene el agente sobre ella es incompleto e impreciso con retrasos en el tiempo. Este entorno es también no determinista [54] porque no siempre se obtiene el mismo resultado cuando se realiza una acción, a veces fallan las ejecuciones dependiendo de las leyes físicas que actúen sobre las entidades en el momento de realizar la acción. También se trata de un entorno estático [54] porque sólo cambia por la interacción de los agentes.

6.1 MODELOS 3D UTILIZADOS

Los agentes se van a desenvolver dentro del escenario mostrado en las siguientes ilustraciones (Ilustración 18, Ilustración 19 e Ilustración 20). El escenario diseñado es muy simple con el fin de poder simplificar los movimientos a implementar y poder centrarnos en la generación de emociones.

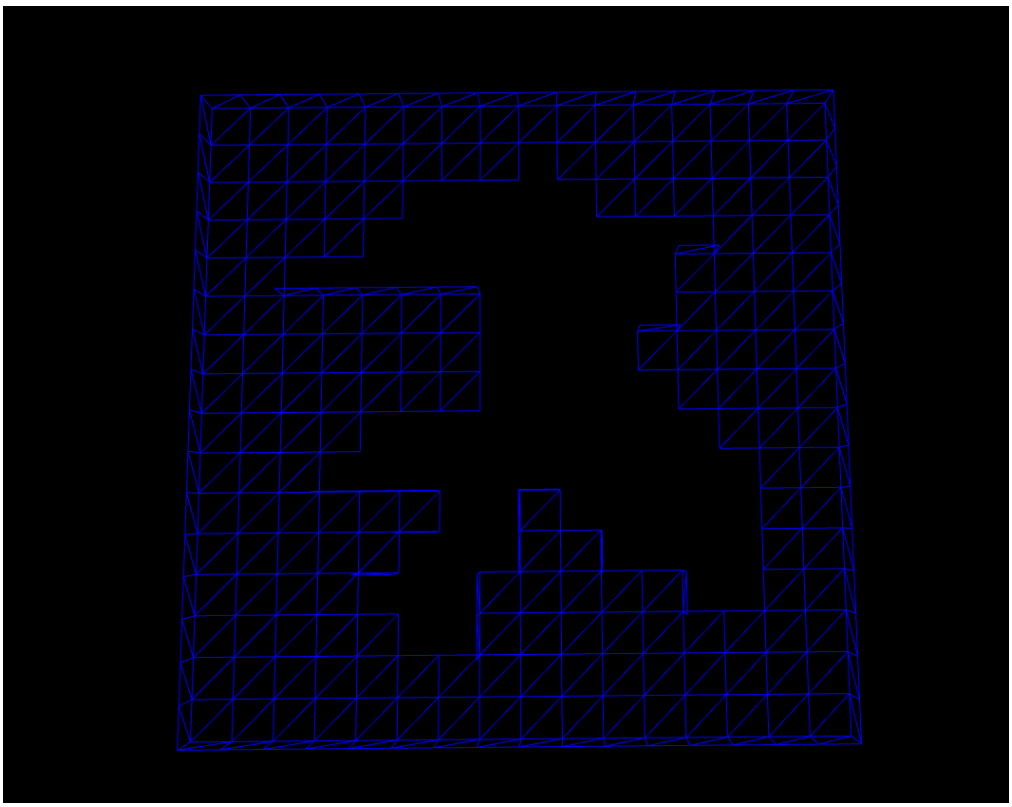

Ilustración 18: el escenario sin texturas visto desde arriba

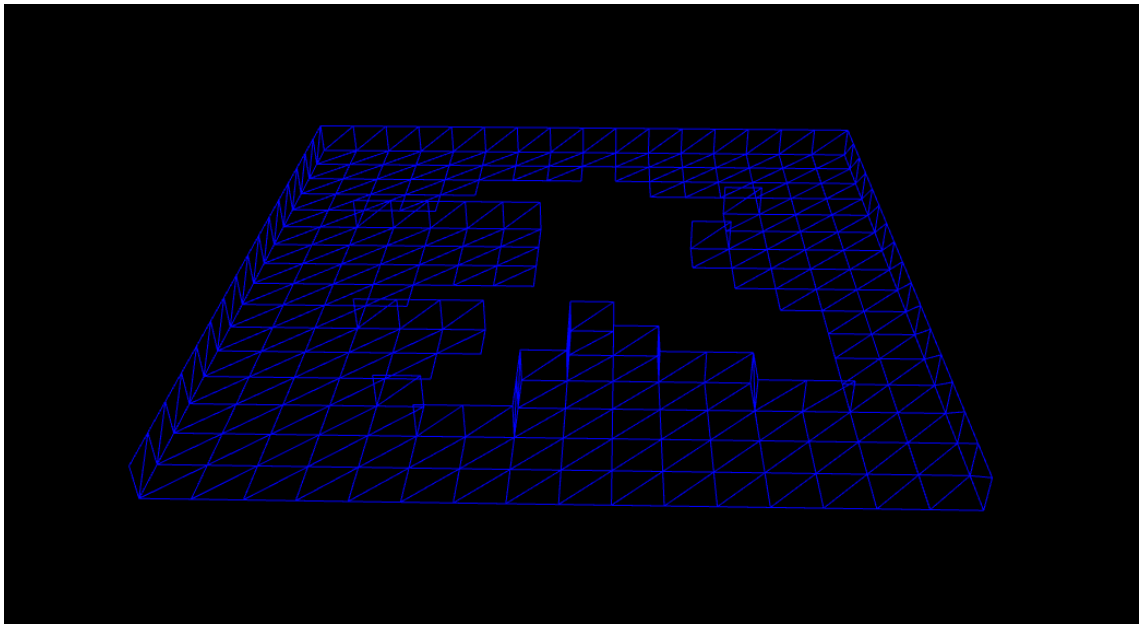

Ilustración 19: escenario sin texturas visto desde otro ángulo

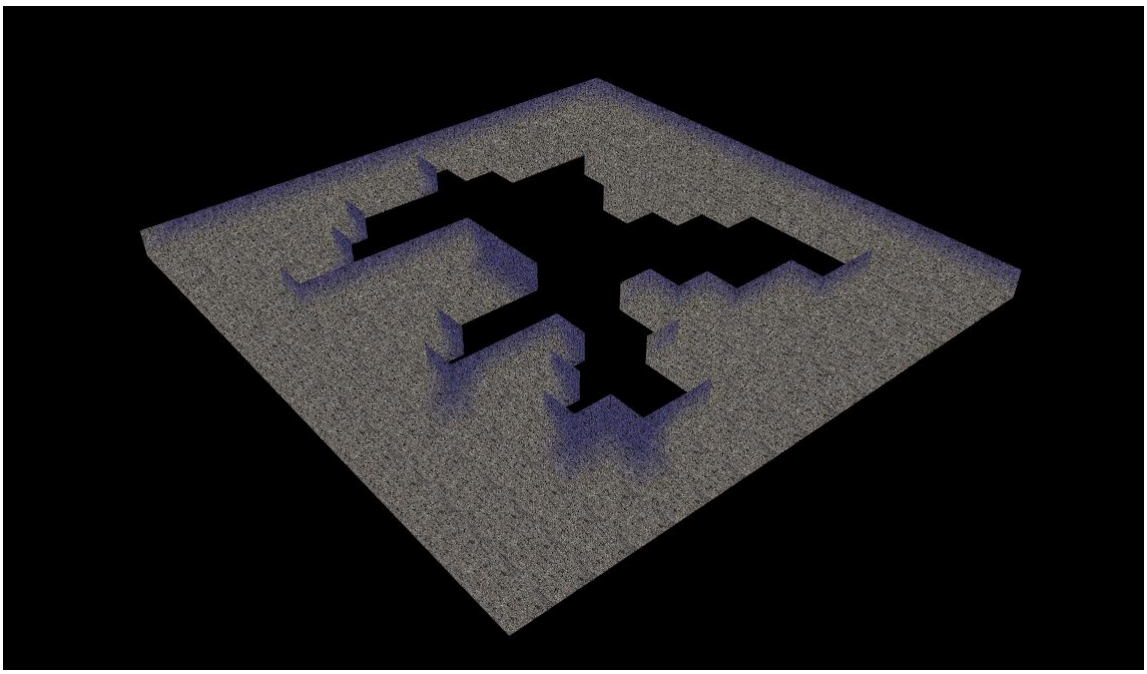

Ilustración 20: el escenario con las texturas

Y el modelo utilizado como "caja" se muestran en la Ilustración 21 y en la Ilustración 22.

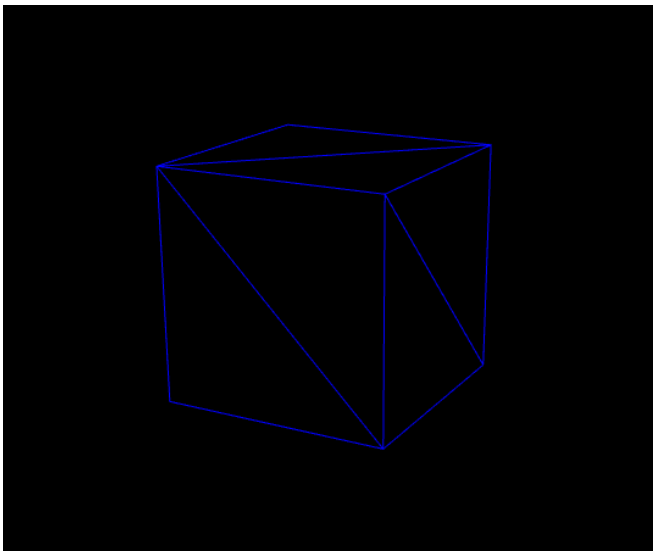

Ilustración 21: la caja sin texturas

Ilustración 22: la caja con las texturas

En la Ilustración 23, Ilustración 24 e Ilustración 25 se muestra el modelo del agente.

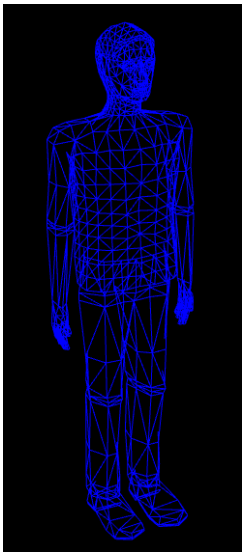

Ilustración 23: el agente sin texturas

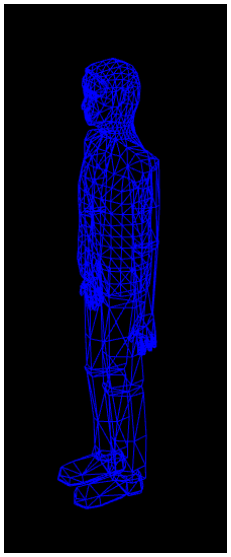

Ilustración 24: el agente sin texturas visto de lado

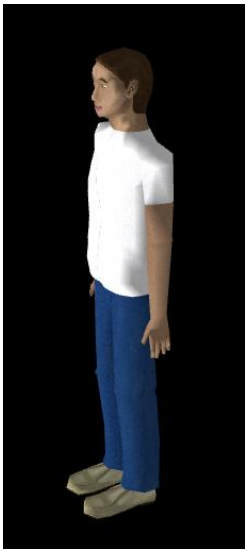

Ilustración 25: el agente con las texturas

Y a continuación se detallan las características de cada emoción básica y los fotogramas de la animación creada para el agente.

6.1.1 ALEGRÍA

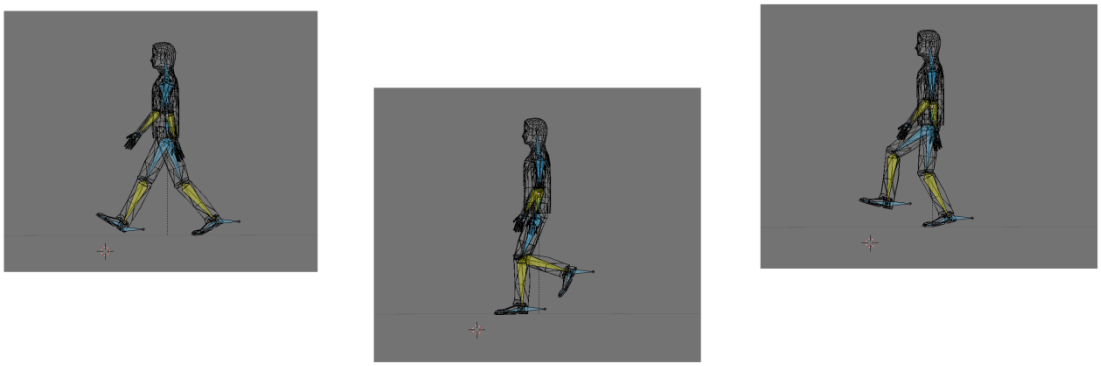

Ilustración 26: animación para la emoción alegría

Cuando el agente está alegre o en estado normal se reproduce la animación de la imagen anterior (Ilustración 26). Tiene la espalda erguida y mira hacia el frente. Es una postura cómoda para él.

6.1.2 TRISTEZA

Ilustración 27: animación para la emoción tristeza

Cuando el agente siente tristeza, encorva la espalda y mira hacia abajo (Ilustración 27). Los movimientos son más lentos, y las piernas y los brazos recorren menos distancia que cuando está alegre. Pierde parte de capacidad para razonar y vaga por el entorno sin perseguir ningún objetivo concreto. Sin embargo, si tenía algún objeto en las manos en el momento de sentir esta emoción, lo sigue sujetando.

6.1.3 IRA

Ilustración 28: animación para la emoción ira

Si el agente siente ira, pone las manos en puño, levanta los hombros y los movimientos de las piernas y los brazos son muy rígidos (Ilustración 28). Sus movimientos son muy acelerados e intimida a los demás.

6.1.4 MIEDO

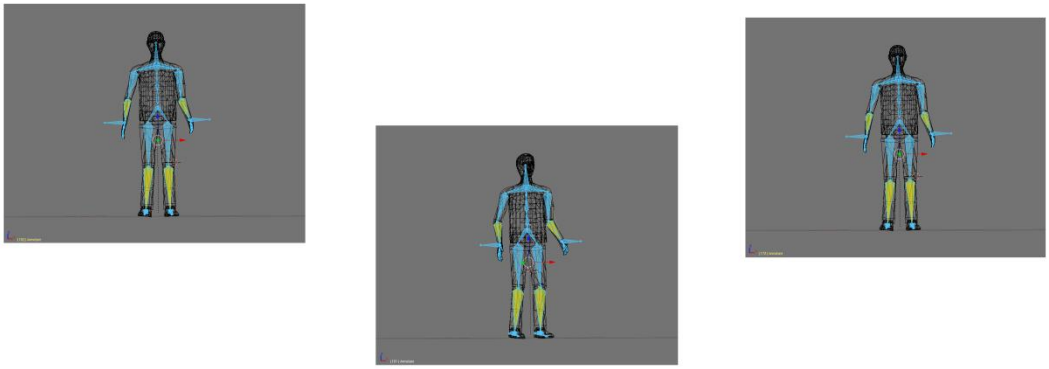

Ilustración 29: animación para la emoción miedo

Al sentir miedo, el agente se queda paralizado y le empiezan a temblar las manos y las piernas (Ilustración 29). Si llevaba algún objeto en sus manos la suelta inconscientemente.

6.1.5 SORPRESA

Ilustración 30: animación para la emoción sorpresa

Ante un estímulo brusco, el agente se queda sorprendido quedándose quieto en el sitio, poniendo las manos hacia delante y echando la cabeza un poco hacia atrás como acto reflejo para intentar alejarse del origen del estímulo (Ilustración 30). Si el agente llevaba algún objeto en las manos antes de sorprenderse, se le caerá de las manos.

6.2 IMPLEMENTACIÓN DEL AGENTE

El siguiente diagrama de secuencia (Ilustración 31) describe la interacción entre el agente y el simulador. Su explicación es análoga a las realizadas en las secciones 5.7 y 5.9, sólo que se sustituye el modelo del sistema genérico por uno concreto que es, el agente.

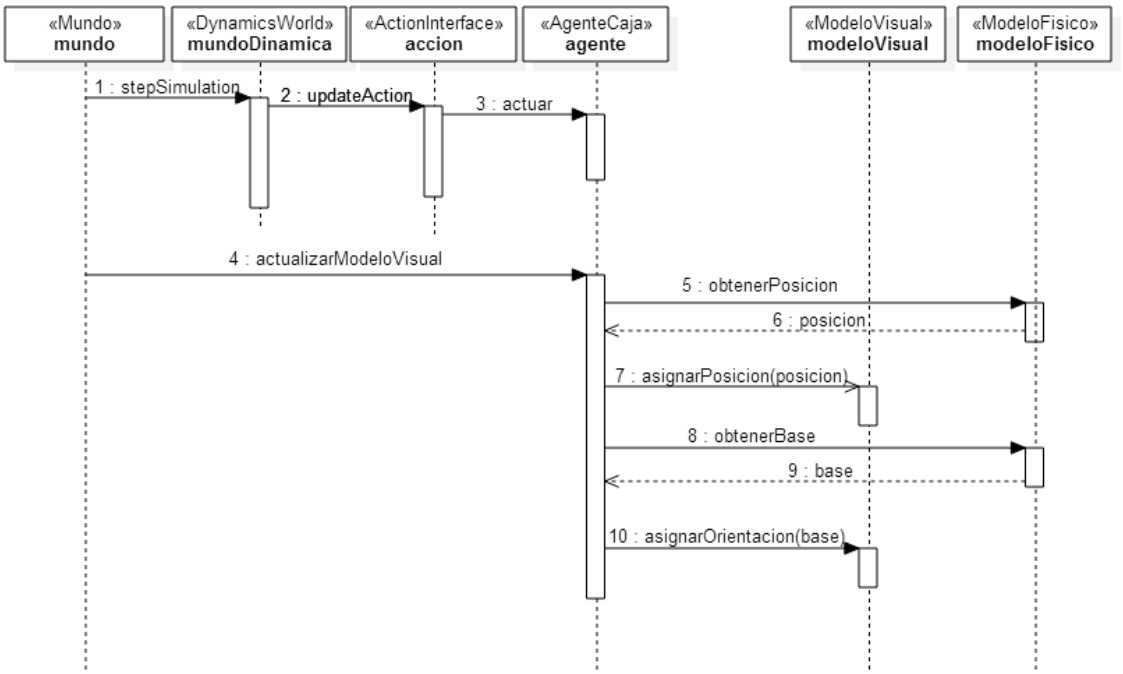

Ilustración 31: secuencia de ejecución de un paso en el agente

El agente que va a interaccionar con el entorno de la aplicación, debe ser capaz de encontrar las cajas con la vista. Para ello, se utilizará una cámara del motor gráfico, que estará situada a la altura de la cara del agente. Éste también deberá ser capaz de esquivar los obstáculos como las paredes, la caja, u otros agentes que se encuentren por el camino. Esto se realiza combinando el planificador de bajo nivel, que detallaremos más adelante, y los mecanismos de detección de colisiones utilizados en el motor físico.

Para la detección de colisiones se han recubierto todos los modelos del entorno con AABB por su eficiencia a la hora calcular los solapamientos entre modelos.

Para el desplazamiento del agente ha sido necesario implementar el método ‘caminar’, pasando como parámetro el desplazamiento. Pero para realizar el desplazamiento hace falta tener en cuenta la altura. Porque si lo movemos sólo sobre el plano XZ, el AABB del agente chocará con los relieves que existan en las zonas del suelo que sean irregulares y no podrá avanzar correctamente. Para añadir el movimiento sobre el eje Y, se ha simulado nuestra forma de caminar (la forma humana): subiendo y bajando las piernas. Realizamos este movimiento en cada invocación al método caminar. Si somos precisos, el número de pasos que damos depende de la distancia que nos desplacemos. Sin embargo, para simplificar, el

agente sólo dará un paso sea la distancia que sea (esta idea se ha sacado de la aplicación de demo que viene en el código fuente de JBullet). Y así sería el movimiento en cada fase (ver Ilustración 32):

1. **Subir pierna**: se desplaza el agente hacia arriba (sobre el eje Y de la coordenada global hacia la dirección positiva) una altura fija ***A,*** que es la altura máxima que puede subir las piernas el agente.
2. **Avanzar**: se desplaza hacia delante (sobre el eje Z de la coordenada local del agente hacia la dirección positiva) la distancia del desplazamiento deseado.
3. **Bajar pierna**: se desplaza hacia abajo (sobre el eje Y de la coordenada global hacia la dirección negativa) la misma distancia ***A*** utilizada en el paso 1, más la aceleración de la gravedad para simular la caída.

En cada uno de estos 3 pasos, se comprueba siempre si colisiona con otros objetos del entorno. De tal forma que, si se produce la colisión, el agente avanza justo la distancia que puede sin traspasar los objetos colisionados. En el siguiente dibujo se muestra una aclaración visual de los 3 pasos descritos anteriormente:

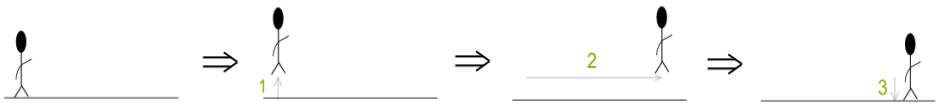

Ilustración 32: desplazamientos realizados al caminar

A la vez que se desplaza el modelo físico del agente, es necesario reproducir la animación de 'caminar' en la malla visual para que parezca una simulación más real. Esto se ha conseguido mapeando el número de *keyframes* a avanzar en la animación con la distancia que recorre el agente. Por cada distancia fija ***D***, se reproduce un ciclo entero de la animación, ya que la animación de 'caminar' es cíclica. Teniendo en cuenta esto, se puede calcular el número de *keyframes* a avanzar de la siguiente forma:

$$Keyframes\ a\ avanzar = round\left(\frac{d}{D} * K_{Tot}\right) \% K_{Tot}$$

donde,

- d es la distancia desplazada
- K_{Tot} es el número de *keyframes* totales en un ciclo completo de animación

6.3 PLANIFICACIÓN DE TAREAS

La planificación de las tareas se va a realizar en dos niveles de detalles: uno de **alto nivel** y otro de más **bajo nivel**.

La planificación de alto nivel se realizará mediante un planificador implementado con Clojure [55] [56], un lenguaje interpretado (también se puede compilar y generar clases Java) con cierta semejanza a la sintaxis de Lisp pero que funciona sobre la máquina virtual de Java. Este es capaz de calcular las secuencias de acciones que permita alcanzar una determinada meta del agente.

Y la planificación a bajo nivel es necesaria para calcular la ruta del agente, y está implementada en Java para permitir una interacción directa con el mundo físico. El mundo físico nos proporciona información acerca de choques entre objetos, y así se puede calcular la ruta del agente evitando colisionar con estos. Se ha diseñado un algoritmo para realizar este cálculo que se detallará más adelante.

6.3.1 PLANIFICADOR DE ALTO NIVEL

El planificador de alto nivel recibe como entrada unas acciones, unos hechos, y unas metas. Y como salida devuelve una lista que contiene todos los resultados encontrados. Cada resultado contiene una secuencia de acciones a ejecutar para llegar a la meta. Y cada acción del resultado está formado por: el nombre de la acción y las instancias concretas que van asociadas en cada variable de la acción.

Como se mencionó anteriormente, la implementación se ha realizado con un lenguaje de scripting llamado Clojure [56]. Se ha elegido este lenguaje por su similitud a Lisp que permite representar y operar fácilmente con listas de predicados, y la interoperabilidad con Java, ya que Clojure funciona directamente sobre su máquina virtual. Esto permite asociar cualquier objeto java en las variables de los predicados, delegando la determinación de la igualdad entre objetos en sus métodos *equals()* y *hashCode()*. Otra característica de Clojure es que no hay problemas de concurrencia al ser un lenguaje funcional y no usa variables compartidas (en su uso habitual), es decir, todos los resultados devueltos por las funciones son instancias nuevas de objetos java en memoria. Esto permite ejecutar varios planificadores dentro de la misma máquina virtual sin preocuparnos por los problemas de concurrencia.

El código fuente del planificador se encuentra en el directorio *src* de la aplicación, concretamente en el fichero *planificador.clj*. Para poder utilizar el planificador dentro del mundo de cajas hay que cargar además el fichero *bloques.clj* que contiene la máquina de estados (FSM [57]) del dominio en forma de acciones y reglas de transición. Este fichero

contiene un mapa con el nombre de la acción como clave y las reglas de transición como valor.

6.3.1.1 Acciones

Cada acción está definida por las precondiciones, los efectos y el nombre. Los hechos y las metas están compuestos por una lista con uno o varios predicados. Un ejemplo de una acción es el siguiente:

```
(def levantar {
  :precondiciones '((en :objeto :lugar))
  :efectos '((en-mano :objeto))
  }
)
```

El nombre de la acción es “levantar” y está asociado con un mapa que contiene las claves: “precondiciones” y “efectos”. Como valor asociado a “precondiciones” tenemos una lista con un predicado: “(en :objeto :lugar)”. Un predicado es una lista que contiene: el nombre del predicado como primer elemento, y cero o más variables. Las variables empiezan por “:” seguido del tipo del elemento que se puede asociar. Estas serán sustituidas, por instancias concretas del elemento tipo indicado, durante el proceso de planificación. Volviendo al predicado del ejemplo, “en” sería el nombre del predicado, “:objeto” una variable de tipo objeto, y “:lugar” sería una variable de tipo lugar. En cuanto a los efectos, tenemos otra lista de predicados. En el ejemplo anterior hay sólo un predicado llamado “en-mano” con una variable de tipo “objeto”.

6.3.1.2 Hechos

Los hechos es una lista de predicados que son verdades en el momento de ejecutar el planificador. Vamos a ver un ejemplo:

```
'((en cajaA PuntoX) (en cajaB PuntoY))
```

En este caso, tenemos una lista de hechos con dos predicados iguales, pero con distintos argumentos. Esto significa que actualmente hay una caja A en el Punto X y otra caja B en el Punto Y.

6.3.1.3 Metas

Las metas, al igual que los hechos, es una lista de predicados. Pero en este caso, son predicados que se desean hacer cumplir mediante la ejecución de las acciones.

6.3.1.4 Resultados

Los resultados que genera el planificador tienen la siguiente forma:

```
(
((despegar {:destino Marte, :objeto1 A, :objeto2 B}))
((despegar {:destino Marte, :objeto1 B, :objeto2 A}))
)
```

Este es el resultado que se obtiene al resolver el siguiente problema:

```
(def cohete {
    :metas '((en A Marte) (en B Marte))
    :hechos '((en Cohete Tierra) (en A Tierra) (en B Tierra))
    :acciones '(despegar)
  }
)
```

Y la definición de la acción “despegar” es la siguiente:

```
(def despegar {
    :precondiciones '((en Cohete :lugar) (en :objeto1 :lugar) (en :objeto2 :lugar))
    :efectos '((en :objeto1 :destino) (en :objeto2 :destino))
  }
)
```

Como se puede observar, al resolver este problema, el planificador devuelve dos resultados. El primero con la A asociada al tipo “objeto1” y la B asociada al tipo “objeto2”. Y el segundo resultado es igual al primero, pero con la asociación inversa. Esto significa que, para conseguir las metas definidas en el problema, basta con ejecutar la acción “despegar” con la primera configuración o la segunda, indistintamente.

6.3.2 PLANIFICADOR DE BAJO NIVEL

El planificador de bajo nivel se encarga de calcular las rutas de desplazamiento del agente realizando pruebas de colisión por el camino. Las pruebas de colisión se realizan dentro del mundo físico, y en este caso en concreto, JBullet permite realizar pruebas de colisión con

haces (trazando una recta desde un punto hacia una dirección) o simulando el movimiento de los objetos de colisión (*CollisionObject*).

Para calcular la ruta del agente, se simula su desplazamiento dentro del mundo físico. En cada simulación se desplaza el cuerpo físico del agente desde la posición inicial hasta la posición objetivo en línea recta. Si en esta trayectoria, el cuerpo físico del agente colisiona con otro cuerpo, el motor físico nos proporciona información relativa a la colisión como pueden ser: el vector normal de la superficie de colisión, el objeto con el que colisiona, instante en el que se produce la colisión, etc. El algoritmo implementado en esta aplicación requiere parte de esta información.

A continuación, se describe un ejemplo de ejecución del algoritmo diseñado. Partimos del escenario inicial descrito por la siguiente ilustración (Ilustración 33):

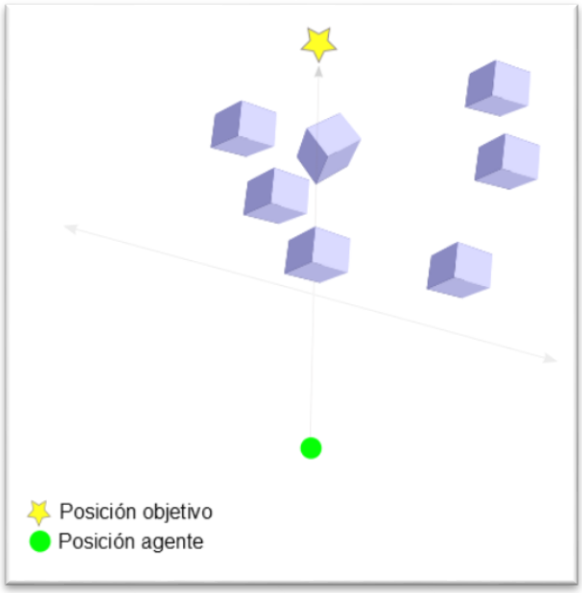

Ilustración 33: entorno con obstáculos

El algoritmo utilizado consiste en simular el desplazamiento de un cuerpo físico en trayectorias en línea recta hasta la posición objetivo, sustituyendo la posición objetivo por una intermedia si se produce una colisión (Ilustración 34). Cada vez que se calcula la posición intermedia se inicia de nuevo el algoritmo partiendo, esta vez, desde la posición intermedia, y se repite el ciclo hasta llegar al objetivo (Ilustración 35) o hasta superar un número máximo de iteraciones. Para más detalle se puede consultar la implementación en *AlcanzarObjetivo.calcularRuta(…)* en el paquete *es.uc3m.simulador3d.aplicacion.modelo.agente.conocimiento.comportamientos*.

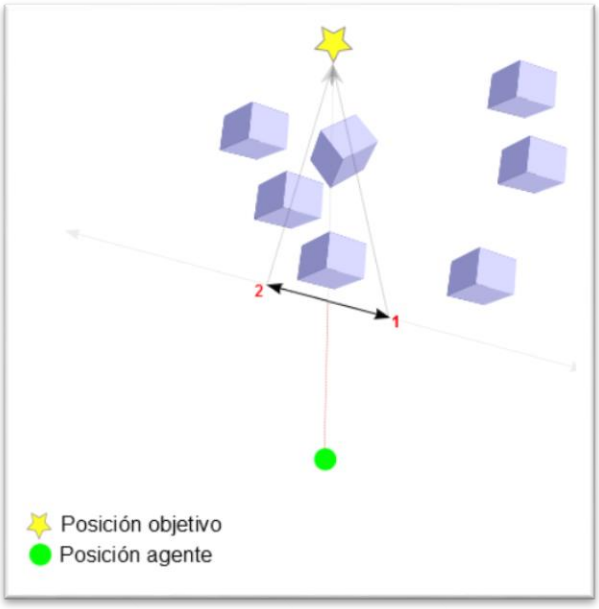

Ilustración 34: resultado parcial tras la primera iteración del algoritmo

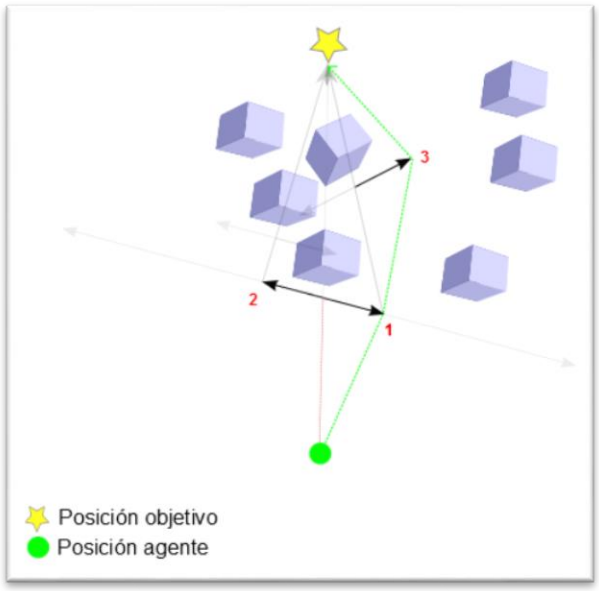

Ilustración 35: resultado final obtenido

6.4 ACCIONES

En este apartado se detallan las acciones que se han implementado en los agentes. Cada acción va relacionada con un paso del plan que el agente lleva a cabo y es la unidad mínima de movimientos que puede realizar.

Las acciones implementadas se pueden observar en el siguiente diagrama de clases (Ilustración 36):

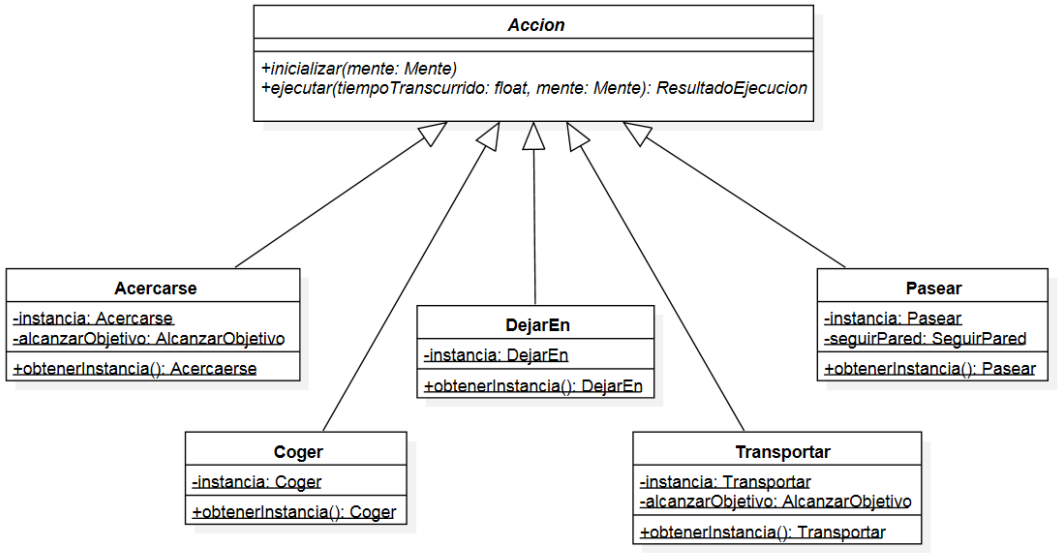

Ilustración 36: diagrama de clases de las acciones implementadas

Una acción se ejecuta con el fin de transitar entre los estados de un plan obtenido con el planificador de alto nivel. Por ello, en cada acción vienen definidas las **precondiciones** que se deben cumplir para poder empezar a ejecutar, los **predicados a eliminar** al completar una acción con éxito, y los **predicados nuevos** a añadir en la lista de hechos al finalizar la acción con éxito.

A continuación, se detallan los predicados definidos, la descripción y los algoritmos implementados en cada una de ellas.

6.4.1 ACERCARSE

Condiciones de transición:

- *Precondiciones*: (en-suelo :lugar :objeto)
- *Predicados a eliminar*: ninguno
- *Predicados nuevos*: (delante-de :objeto)

Descripción: Mira hacia la posición de un objeto y va caminando hasta llegar a una distancia que sea "alcanzable" al objeto.

Algoritmo:

- Si ve el objeto,
    - Pero lo tiene cogido otro agente, se resigna y da por fallida la acción
    - Si sigue en el suelo, se acerca hasta que el objeto sea "alcanzable" por el agente. En cuanto el objeto sea alcanzable, termina la acción con éxito.
- Si no ve el objeto, espera 5 segundos.
    - Si a los 5 segundos sigue sin verlo se resigna y la acción termina como fallida.
    - Si vuelve a ver el objeto antes de los 5 segundos, sigue la ejecución de la acción.

6.4.2 COGER

Condiciones de transición:

- *Precondiciones*: (manos-libres), (en-suelo :lugar :objeto), (delante-de :objeto)
- *Predicados a eliminar*: (manos-libres), (en-suelo :lugar :objeto), (delante-de :objeto)
- *Predicados nuevos*: (en-mano :objeto)

Descripción: Coge un objeto con las manos.

Algoritmo:

- Si no es alcanzable, o no ve el objeto, o lo tiene otro agente, se resigna y da por fallida la acción
- Si no se cumple lo anterior, ejecuta el siguiente algoritmo:
    - Si no tiene el objeto sujetado y:
        - Si está de pie, empieza a agacharse
        - Si está agachado, sujeta el objeto
            - Si no pudo sujetar el objeto, da la acción por fallida
    - Si tiene el objeto sujetado:
        - Si está agachado, se pone de pie
        - Si está de pie, da la acción por finalizada con éxito

* Condiciones de es alcanzable: si la distancia al objeto es menor o igual a 2.8 unidades y el objeto está en un ángulo inferior a 0.1 radianes con respecto al vector frente del agente.

6.4.3 DEJAREN

Condiciones de transición:

- *Precondiciones*: (en-mano :objeto), (situado-en-lugar-de-colocacion)
- *Predicados a eliminar*: (en-mano :objeto), (situado-en-lugar-de-colocacion)
- *Predicados nuevos*: (manos-libres), (en-suelo :lugar :objeto), (colocado :objeto)

Descripción: Se agacha para dejar el objeto en el lugar elegido.

Algoritmo:

- Si está de pie se agacha
- Si está agachado y
    - Tiene la caja sujetada, la suelta
    - No tiene la caja sujetada, se pone de pie y da la acción por terminada con éxito

6.4.4 PASEAR

Condiciones de transición:

- *Precondiciones*: ninguno
- *Predicados a eliminar*: ninguno
- *Predicados nuevos*: ninguno

Descripción: Se mueve por el entorno para percibir estímulos.

Algoritmo:

- Camina hasta chocar
- Si ha chocado, ejecuta el comportamiento "*SeguirPared*"

* Durante la ejecución de esta acción, si el agente percibe un estímulo visual de una caja, puede desencadenar la selección de una meta en su mente y posteriormente, la elaboración y ejecución de un plan para conseguir dicha meta. La acción no se llama "Explorar"

porque no es una acción proactiva, sino reactiva; no hay ningún procesamiento de alto nivel, es decir, no utiliza la función ejecutiva del lóbulo frontal.

6.4.5 TRANSPORTAR

Condiciones de transición:

- *Precondiciones*: (en-mano :objeto)
- *Predicados a eliminar*: ninguno
- *Predicados nuevos*: (situado-en-lugar-de-colocacion)

Descripción: Con el objeto sujetado, busca el lugar de colocación y se sitúa en él.

Algoritmo:

- Si no tiene el objeto sujetado, da la acción por fallida
- Si tiene el objeto sujetado, calcula la distancia entre la posición actual del agente, y la posición de colocación
    - Si la distancia es mayor a 10 unidades, ejecuta el comportamiento "*AlcanzarObjetivo*" para disminuir la distancia al lugar de colocación
    - Si la distancia es menor o igual a 10 unidades, se considera que está muy cerca del lugar de colocación y da por terminada la acción con éxito

6.5 COMPORTAMIENTOS

Los comportamientos que se definen a continuación son rutinas de movimientos auxiliares que permiten realizar algunas de las acciones anteriores.

6.5.1 ALCANZAROBJETIVO

Descripción: Intenta colocarse al lado de un objeto o un lugar.

Algoritmo:

- Obtiene la posición del objetivo (objeto o lugar)
- Si no hay una ruta calculada, o si la posición es distinta al ciclo anterior, se calcula la ruta utilizando el ***planificador de bajo nivel***
    - Si ha encontrado una ruta, se desplaza siguiendo la ruta. Si la distancia entre el agente y el objetivo es menor a 1 unidad, o ha chocado con el objeto objetivo, se considera que se ha alcanzado el objetivo.

- Si no ha encontrado una ruta, ejecuta el comportamiento "*SeguirPared*" durante 1 segundo.

6.5.2 SEGUIRPARED

Descripción: Avanza siguiendo la pared.

Algoritmo:

- Simula el desplazamiento hacia el frente
  - Si choca y,
    - Si no estaba girando antes, gira hacia el lado que haga girar menos cantidad de ángulo, utilizando el vector normal del punto de colisión en la pared.
    - Si estaba girando antes, sigue girando en la misma dirección.
  - Si no choca,
    - Camina siguiendo la pared que hay en el lado opuesto a la dirección de giro utilizada en el paso anterior.
    - Si la pared desaparece en el lado indicado, gira hacia dicho lado utilizando como pivote la esquina de la pared hasta encontrar una pared que seguir.
    - Si colisiona de frente, gira siguiendo la dirección de giro hasta que la pared quede en el lado opuesto a la dirección de giro.

6.6 INTERACCIÓN ENTRE LAS CLASES QUE COMPONEN EL AGENTE

Al ser invocado el método *AgenteCaja.estimular()* por la clase *MundoConcreto* en cada ciclo de ejecución del bucle principal, se invoca al método estimular de todos los agentes registrados en el entorno. Todos los estímulos que se notifican son visuales (del tipo *EstimuloVisual*) y se notifica uno por cada objeto visto del entorno. Estos estímulos se encolan dentro de la memoria del agente y al terminar de notificar todos se procesan invocando al método *Mente.procesarEstimulos(estimulos)*. En el procesamiento, primero pasan los estímulos por la zona de la memoria llamada memoria implícita que se encarga de procesar los estímulos en 'crudo' y alerta al agente si es necesario prestar atención a uno o varios de ellos, devolviendo un valor mayor de cero. Este valor es calculado y devuelto por la zona 'no consciente' de la mente que alerta inmediatamente a la parte consciente si hay

alguna posible amenaza (el algoritmo utilizado se detalla más adelante). Este valor es el nivel de atención que presta el agente a los estímulos. Posteriormente, se calcula la valencia asociada al plan y el paso en ejecución en la memoria emocional, donde se encuentran los estímulos que hicieron emerger una emoción asociada al contexto. Remarcar que este estímulo también viene de la parte 'no consciente' de la mente y hace presentir al agente de algo bueno o malo según la situación. Esto es porque la mente le da mayor importancia a los hechos o situaciones en los que uno tiene una experiencia emocional y lo registra en la memoria emocional (esto se refleja en los experimentos de LeDoux y Damasio).

Después, la mente del agente pasa a evaluar, también de forma 'no consciente', la expansión temporal de sus acciones utilizando una parte de memoria que se ha llamado 'memoria temporal', que se encarga de calcular y almacenar los tiempos 'medios' de ejecución de cada una de sus acciones en el pasado. Sirve para indicar a la parte consciente de la mente de que algo inusual ocurre cuando hay retrasos o adelantos con respecto al tiempo 'medio' almacenado. Esta zona de memoria nos permite simular la producción de fatiga en el agente. Mientras que el tiempo de ejecución de una acción no sea superior al tiempo 'medio', la excitación (*arousal*) del agente va subiendo poco a poco para poder centrarse en la ejecución de la acción. Sin embargo, al pasar del tiempo 'medio', la excitación empieza a bajar porque ha llegado al límite de su concentración. Este efecto se ve en la *curva de trabajo* de Emil Kraepelin [58] y como el momento en el que disminuye la concentración no está claro, se ha utilizado el tiempo 'medio' de ejecución de las acciones para utilizarlo como el momento en el que empieza a manifestarse la fatiga.

Una vez evaluados los puntos anteriores, aquí interviene la parte consciente de la mente, sólo si: el valor de la 'atención', obtenido utilizando la memoria implícita, dio un número mayor que cero. Porque esto indica la existencia de alguna posible amenaza, el cual requiere la intervención de la parte consciente para identificar si realmente es o no una amenaza. En el caso de que la atención no sea cero, el agente debería evaluar la situación para intentar evitar posibles situaciones desfavorables para él. Sin embargo, para enfocar este trabajo a las manifestaciones de emociones básicas, se omite este paso y simplemente la mente accede a la memoria de trabajo para notificar los estímulos visuales. Dicho de otra forma, si la memoria implícita no activa la atención, el agente no interpreta los estímulos visuales y no estaría viendo nada en la práctica, aunque los estímulos estén entrando por sus ojos.

Una vez procesadas las 3 fases anteriores, se obtiene la valencia y la excitación que hay que aumentar o disminuir en el organismo del agente. La valencia se obtiene de la memoria emocional si existe algún recuerdo del pasado asociado con el contexto de ejecución. Y la excitación corresponde a la atención obtenida en la memoria implícita, más el valor obtenido

de la memoria temporal. Estos valores se notifican internamente al organismo en forma de estímulo fisiológico que hace desplazar el punto PAD dentro del agente. Los estímulos fisiológicos sólo desplazan el punto sobre los ejes P y A, ya que la sensación de dominio (el eje D) lo determina la parte consciente de la mente y éste es desplazado durante la ejecución de acciones deliberadas del agente.

Una vez emitido el estímulo al organismo, termina el procesamiento de los estímulos evaluando la posición actual del punto PAD para comprobar si del agente emerge alguna emoción básica. En el caso de que emerja alguna emoción, interviene la memoria emocional y registra la valencia actual asociado con el contexto actual de ejecución para ser utilizado en posteriores ciclos de ejecución.

A partir de este punto, vamos a detallar la interacción entre los distintos componentes que compone el agente. Este es el diagrama con las clases participantes (Ilustración 37):

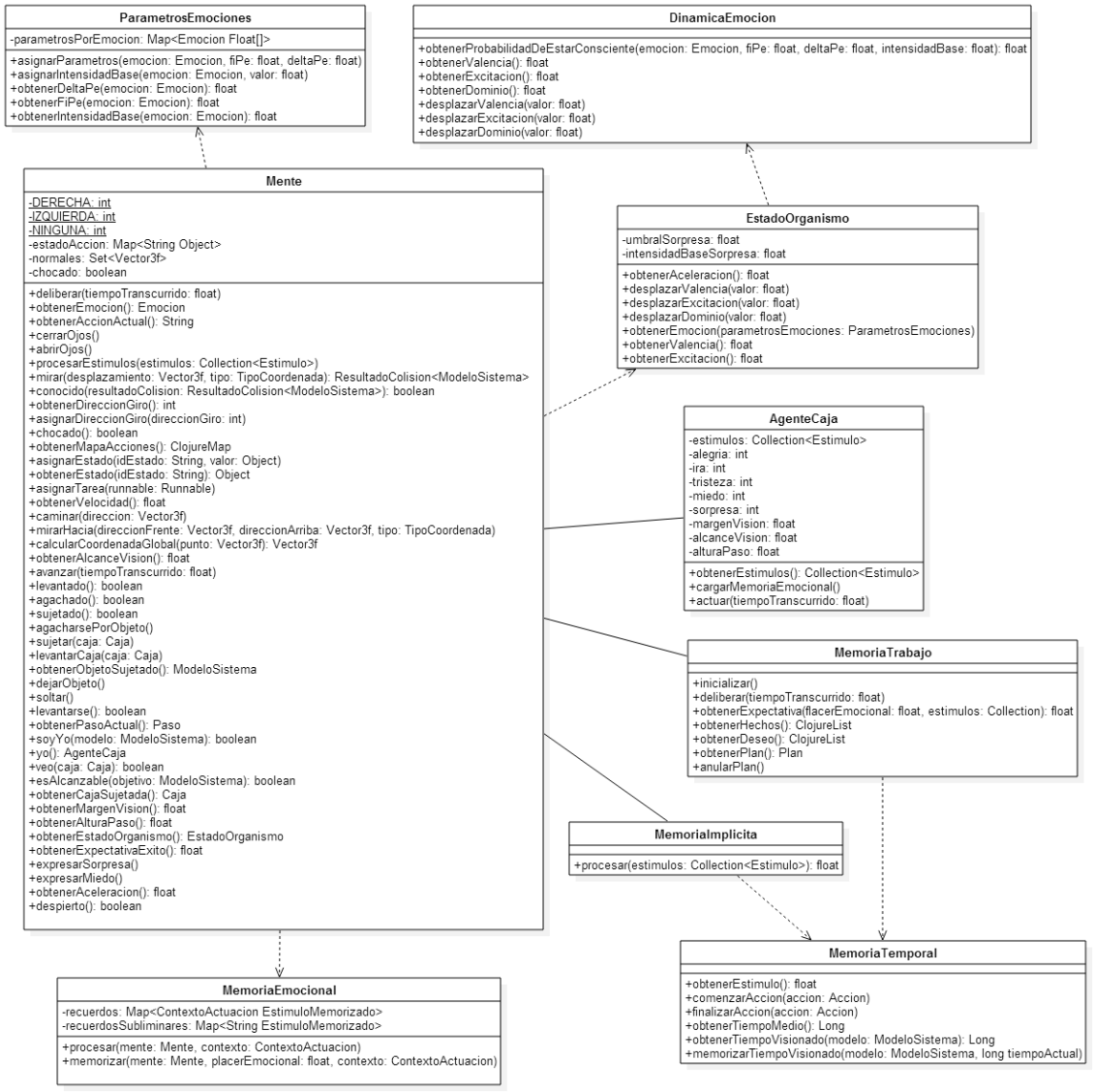

Ilustración 37: dependencias entre las clases que compone el agente

Cuando el agente necesita realizar una intervención consciente, se ejecuta el método *AgenteCaja.actuar(tiempoTranscurrido)*. Este método desencadena la ejecución deliberada por el agente haciendo uso del planificador, salvo en dos casos excepcionales que se detalla más adelante. Antes de actuar, el agente comprueba si se ha manifestado alguna emoción. Si no hay ninguna emoción emergente, el agente delibera. Las dos excepciones que se han comentado anteriormente son:

1. Cuando la mente no está despierta, es decir cuando la excitación del organismo es cero. Indica que el agente está inconsciente.
2. Cuando la emoción manifestada es la tristeza. En este caso, el agente vaga por el entorno sin perseguir su objetivo. El agente avanza esquivando paredes y obstáculos, abandonando la ejecución del plan, de forma temporal, si se estaba ejecutando alguno. Pero si el agente llevaba una caja en las manos, sigue agarrándola. Es una forma de representar un conflicto entre la parte consciente y la no consciente de la mente que se manifiesta en estas situaciones.

Durante la ejecución de este método, se ejecutan los siguientes pasos:

1. Se comprueba la emoción que está sintiendo el agente.
2. Y si lo que siente es SORPRESA, se ejecuta la animación de sorprendido, y si tenía sujetada alguna caja, la soltará por el impacto del estímulo o estímulos.
3. En el caso de sentir MIEDO, se ejecuta la animación miedo, e igual que en el caso anterior, suelta la caja en el caso de que estuviera sujetando una.
4. Si la emoción sentida es TRISTEZA, ejecuta el método *Mente.avanzar()* que hace que el agente camine recto esquivando obstáculos y bordeando las paredes, pero sin soltar la caja, si estaba sujetando una.
5. En el resto de los casos, el agente delibera para pasear o ejecutar un plan.

Para deliberar, el agente invoca el método: *Mente.deliberar()*. Al ejecutar este método, el agente accede a la memoria de trabajo porque es una acción que se interactúa con la consciencia. Desde ella, el agente invoca al planificador de alto nivel para pensar. Si acaba de percibir un estímulo nuevo, utiliza este planificador para buscar algún plan que permita cumplir su deseo. Si no tiene ningún deseo en mente, pasea por el entorno en busca de nuevos estímulos. Si el agente ya tenía un deseo en mente, intenta componer un plan que permita cumplirlo. Cuando el agente "piensa" en este plan, se procesa en un hilo aparte para no bloquear el bucle principal del motor gráfico. Este proceso puede durar varios ciclos de ejecución del simulador, por lo que el agente no realizará ninguna acción mientras que la

ejecución del planificador esté en marcha. Si encuentra algún plan, la ejecuta. Y en el caso de que no encuentre ninguna solución, percibe un estímulo fisiológico negativo como signo de frustración.

Cuando se activa la alerta desde la zona de memoria implícita, primero se registran los objetos vistos añadiendo el predicado (visible “modeloVisto”) en la lista de hechos de la memoria de trabajo del agente para que el planificador de alto nivel lo tenga en cuenta.

En el caso de que haya visto una caja pueden ocurrir tres cosas:

1. Si no tiene ningún deseo en mente, y si no está colocada, selecciona de la lista de deseos sin cumplir, el deseo de colocar la caja para llevarlo a cabo inmediatamente. En el caso de que no esté en la lista de deseos y esté en la lista de los deseos cumplidos, esto indicaría que la caja estaba colocada la última vez que la vio. Sin embargo, en el momento actual se ha movido por alguna razón, posiblemente porque otro agente lo haya movido. En este caso el agente siente que ha perdido el dominio de la situación y disminuye el dominio.
2. Si ya está ejecutando otro plan, para cumplir otro deseo ajeno a la caja, la ignora. Pero al igual que en el caso anterior, si observa que la caja ha sido movida de su lugar de colocación, disminuye el dominio.
3. Si no tiene ningún deseo en mente, y ya está en la lista de deseos cumplidos. Se activa el mecanismo para mantener su estado. En esta situación, aumenta el dominio si la caja sigue en su sitio. Sin embargo, este disminuye si lo ve en manos de otro agente, y además percibiría un estímulo negativo (valencia con intensidad negativa).

El siguiente diagrama de clases (Ilustración 38) se describen las relaciones de dependencias entre los componentes principales que interaccionan cuando se hace uso del planificador de alto nivel.

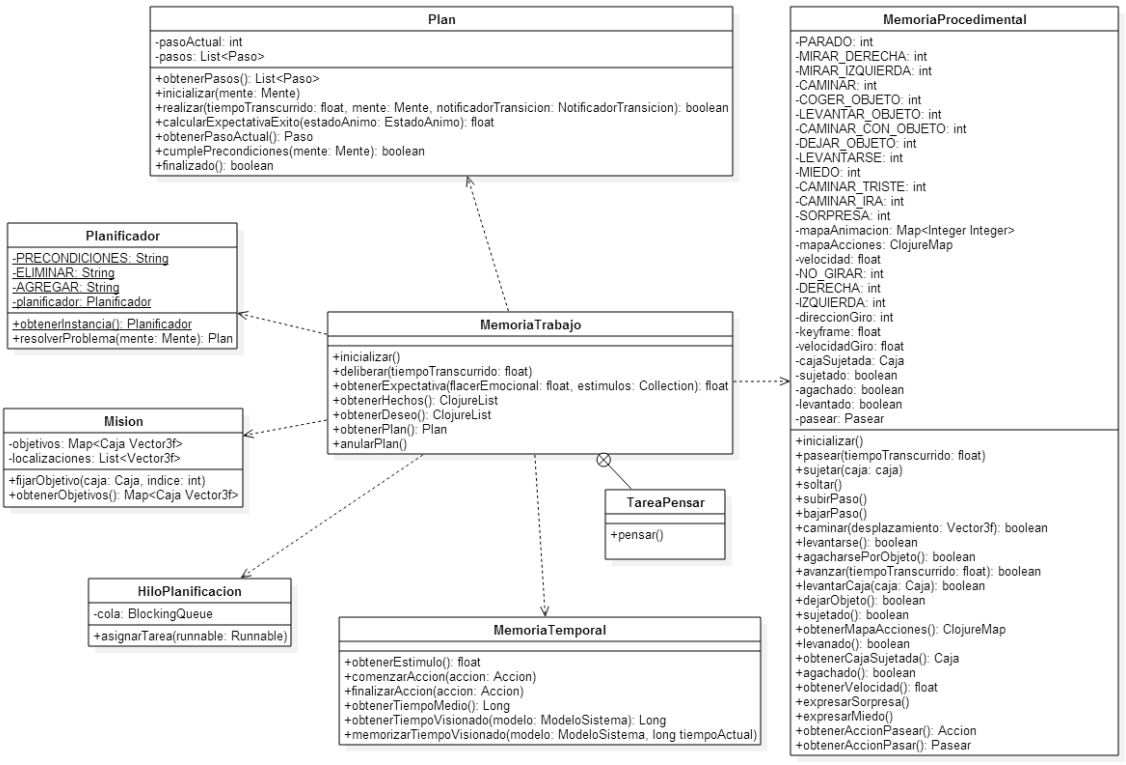

Ilustración 38: dependencias de la memoria de trabajo

Al ejecutar el planificador de alto nivel se invoca el método *TareaPensar.pensar()*. Como se comentó anteriormente, *e*ste método se ejecuta en otro hilo independiente para no bloquear el bucle principal del motor gráfico. Primero, se invoca al método *Planificador.resolverProblema(Mente)* para intentar componer un plan. En el caso de que el método anterior haya devuelto un plan, se inicializa este plan para poder ejecutar sus pasos. Y en el caso contrario, se marca como plan no encontrado.

Concretando más, el método *Planificador.resolverProblema(Mente),* intenta generar un plan que permita cumplir el deseo actual utilizando el planificador de alto nivel. Para ello, obtiene los predicados del deseo (meta a conseguir), y los predicados de hechos. Después, invoca al planificador implementado en Clojure. En el caso de que no se obtenga ningún resultado, devuelve *null*. Y si hay resultados, se transforman los objetos de Clojure obtenidos, en una instancia de tipo *Plan*.

Una vez que se compone un plan, el agente empieza su ejecución a partir del siguiente ciclo de ejecución. Para ello, invoca el método *Plan.realizar()* que se encarga de ejecutar los pasos que compone el *Plan* de forma ordenada. Este método llama al método *Paso.ejecutar()* del paso actual y comprueba su resultado. Si el resultado de ejecución es EXITO, obtiene el siguiente paso, y añade un valor positivo de dominio y devuelve *true*. Si el resultado es FALLO, se genera un estímulo negativo en función del nivel de excitación que tenía el agente en el momento, y *Plan.realizar()* devuelve *false*. Y en el caso de que el resultado sea

SIN_TERMINAR, no se hace nada adicional y devuelve *true*. En el siguiente diagrama de clases (Ilustración 39) se puede observar la relación entre las clases que conforman un plan.

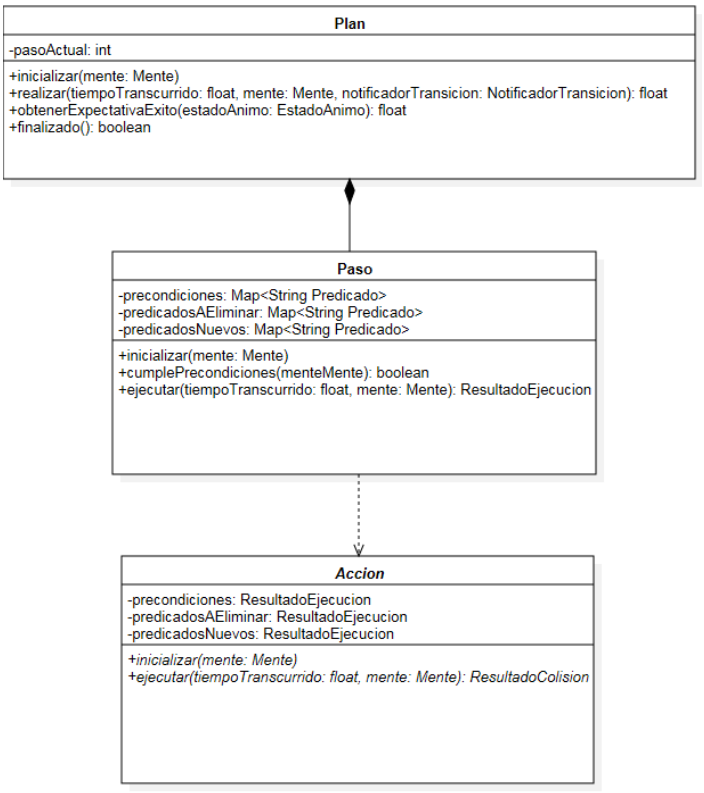

Ilustración 39: composición de un plan

6.6.1 SIMULACIÓN DE EMOCIONES

En el agente se ha implementado lo que hemos interpretado de las teorías de LeDoux y Damasio. Para simplificar el problema, se ha omitido la implementación de la memoria consciente del agente. Ya que esto no es necesario para la generación de emociones básicas, que es el principal objetivo de este documento. Si se implementara la memoria consciente (zona de memoria accesible por la mente), implicaría diseñar una estrategia de toma de decisiones basada en las dos memorias, la consciente y la no consciente. Dado que las emociones básicas producen unos efectos inmediatos tanto en el interior como en el exterior de un ser vivo; con reproducir este hecho consideramos que se consigue el objetivo de este proyecto. Por ello, se implementará sólo la memoria no consciente.

El marcador somático asocia las entidades que han causado los estímulos emocionales con su valencia percibida. Para que cuando el individuo vuelva a encontrarse con dichas entidades, el mecanismo del marcador somático active una serie de estímulos con valencia positiva o negativa basándose en las emociones percibidas en el pasado. Pero sin que la consciencia intervenga en el proceso. Pensamos que tiene una analogía con las 'heurísticas' de un algoritmo de búsqueda. El marcador hace de heurística y lleva asociado un comportamiento adecuado para maximizar la probabilidad de supervivencia del individuo. Un comportamiento pre-definido en el cerebro a través de los genes que ha ido evolucionando generación tras generación y que sigue evolucionando. Si, por ejemplo, si nos encontramos por primera vez con un gato de apariencia inofensivo, no nos asustamos; pero si ese gato nos ataca, es probable que sintamos miedo de ese gato. Y la próxima vez que el gato se cruce en nuestros caminos, se activan los mecanismos del marcador somático y el pulso se acelera y nuestra consciencia se dará cuenta de ello y nos producirá una sensación negativa (estímulo interno de valencia negativa). En situaciones extremas puede que hayamos cogido fobia a los gatos por esta experiencia y la mala sensación sea habitual al encontrarnos con cualquier gato. Cuando hablamos de memoria no consciente, nos referimos a esta 'heurística' aprendida de forma inconsciente, sólo a través de los estímulos percibidos y las emociones generadas.

En el algoritmo utilizado en el agente, existen dos formas de percibir las entradas de tipo valencia:

1. Mediante el proceso de evaluación del plan elegido. En este caso, la elección del plan es totalmente arbitraria, no está influenciada, en absoluto, por la consciencia y/o la no consciencia. Esto es para simplificar la implementación y no desviar el foco, de las emociones básicas.

2. Mediante las experiencias almacenadas en el marcador somático y su manifestación como reacción a los estímulos percibidos por las entidades que se proyectan en nuestra visión. También pueden afectar los estímulos que genere nuestro propio cuerpo de forma interna, pero como no vamos a simular el comportamiento de todo el cuerpo como pueden ser órganos, músculos, sistema nervioso, venas, arterias, etc.; los vamos a agrupar en un concepto que nos proporcione una valencia del organismo: el ***estímulo fisiológico***.

El *arousal*, interpretamos que es la capacidad de focalizar la atención hacia el ente o entes que nos pueda influenciar positiva o negativamente. Pensamos que la atención se centra siempre en los cambios del entorno. Cuanto más brusco sea el cambio, mayor será la atención que prestaremos a los factores que hayan producido ese cambio; y cuanto menos sea el cambio menos atención se prestará al entorno en cuestión. Para ello se ha centrado en la visión del agente, ya que la visión tiene un gran peso a la hora de evaluar nuestra interacción con el entorno que nos rodea. Y omitimos la evaluación de los estímulos táctiles a la hora de calcular las emociones, también por simplificar la implementación.

6.6.1.1 Dimensión Pleasure

Esta dimensión representa el placer (o la valencia, que es el término utilizado en este documento) de los estímulos que percibe el agente. Determina el grado de bondad o maldad de los estímulos percibidos. Considerando que la cualidad de bondad o maldad, es totalmente subjetiva. Es decir, cada individuo interpreta los estímulos según la ganancia o pérdida que pueda suponer en su organismo. Esta dimensión puede tomar tanto valores positivos como negativos. Considerando como un estímulo malo los valores negativos, y los positivos como buenos.

En la aplicación se utiliza el rango -1.0 hasta +1.0 en números reales. Cuanto más cerca esté de conseguir una meta, mayor será la cantidad que se aporta en esta dimensión; y cuando más se aleje de conseguir una meta, mayor será la cantidad que se resta en esta dimensión. Si en algún momento el valor en esta dimensión sobrepasa los límites de su rango, será truncado en -1.0 o +1.0 dependiendo del extremo sobrepasado.

6.6.1.2 Dimensión Arousal

El eje *arousal* representa el grado de excitación o alerta. Su rango va desde 0.0 hasta +1.0 y está representado mediante números reales. Cuanto mayor sea este valor, mayor será el grado de concentración del agente sobre el estímulo o los estímulos que esté percibiendo.

Por el contrario, si este valor vale 0, indica que el agente está inconsciente, dormido, o en el peor de los casos, muerto.

Para el mundo de cajas, al ponerse el *arousal* a 0 (< 0.001) el agente se pone a descansar para recuperar el *arousal* para volver a estar activo. Cuando el agente se pone a descansar se queda parado en el lugar durante 1000 ciclos de ejecución para recuperar el *arousal*. Al terminar los 1000 ciclos se pone el *arousal* a 0.8, la valencia a 0.0, y el dominio a 0.0. Y también se resta 7 horas a los tiempos de inicio registrados en la memoria temporal para cada estímulo contextual memorizado. Esto es para simular un sueño de 7 horas. Al terminar el descanso, el agente despierta con un estado del organismo equilibrado y vuelve a perseguir sus objetivos.

Al igual que la dimensión anterior, si sobrepasa alguno de los límites será truncado al valor del límite más cercano.

6.6.1.3 Dimensión Dominance

Esta dimensión representa el grado de dominio o sumisión que considera el agente que siente ante su situación actual. Su rango está definido entre -1.0 y +1.0 en números reales. A mayor dominio de la situación, mayor será su valor; y cuanto más sumiso se considere, su valor estará más cerca del -1.0.

En una primera implementación se consideró no utilizar esta dimensión, sin embargo, al empezar a implementarlo nos dimos cuenta de que sin esta dimensión no se podía distinguir entre la emoción de ira y el miedo. Ya que desde el punto de vista de las otras 2 dimensiones se sitúan en una región idéntica. Al añadir la tercera dimensión se pudo concluir que la ira se manifiesta cuando el agente siente que tiene la situación bajo control (sensación de dominio), y el miedo se manifiesta cuando siente que no tiene el control de la situación (sensación de sumisión).

6.6.2 GENERACIÓN DE EMOCIONES

En la siguiente tabla (Tabla 1) están definidos los distintos puntos en el espacio PAD para cada una de las emociones básicas. La gran parte de estos puntos están extraídos del documento [35] pero en vez de ir de 0 a 100, va de 0.0 a 1.0 para facilitar los cálculos en los que interviene el *arousal*, y en vez de números enteros, con números reales.

Emoción básica	Valencia(P)	Arousal(A)	Dominance(D)
ALEGRIA	0.5	0.2	1.0
ALEGRIA	0.5	0.2	-1.0
ALEGRIA	0.8	0.8	1.0
ALEGRIA	0.8	0.8	-1.0

MIEDO	-0.8	0.8	-1.0
IRA	-0.8	0.8	1.0
SORPRESA	0.1	0.8	1.0
SORPRESA	0.1	0.8	-1.0
TRISTEZA	-0.5	0.2	1.0
TRISTEZA	-0.5	0.2	-1.0

Tabla 1: definición de los puntos PAD de cada emoción

Una vez definidos los puntos en el espacio PAD, pasamos a explicar su dinámica. Las clases involucradas en la generación de las distintas emociones básicas son las clases *EstadoOrganismo* y *DinamicaEmocion*. La primera clase se encarga de gestionar el punto PAD dentro del organismo. Y la segunda clase se encarga de calcular la probabilidad de que el agente sea consciente de estar experimentando una emoción.

La clase *EstadoOrganismo*, como se acaba de mencionar, gestiona el punto PAD del organismo del agente. Además, permite obtener los valores actuales de los componentes P, A y D del agente a través de ella. Y también se encarga de estas cosas:

1. Se encarga de registrar el nivel de aceleración del organismo, que varía según el *arousal* del agente. Cuanto más grande sea el *arousal* más acelerado estará el agente; y cuanto más pequeño, menos acelerado se encontrará el agente.
2. Se encarga de mantener el equilibrio emocional del agente desplazando los componentes P y A para simular el proceso de homeostasis. La velocidad con la que se alcanza el equilibrio del organismo varía según la aceleración del organismo mencionado en el punto anterior. En este proceso no se ve involucrado el componente D porque es un componente totalmente subjetivo y consciente del agente.
3. Se encarga de activar la intensidad base para la emoción SORPRESA, comprobando si la variación de excitación supera o no, el umbral de sorpresa (U_s).
4. Calcula la emoción del agente:
    - Calcula la probabilidad para cada emoción invocando a la clase *DinamicaEmocion*. En el caso de SORPRESA, se tiene en cuenta la intensidad base de sorpresa calculada usando el mecanismo del punto 3.
    - Si existe 1 o más emociones con probabilidad distinta de 0, se activa la emoción con mayor probabilidad. Si no existe, devuelve ‘null’ que indica que no hay ninguna emoción activada.

Para calcular y hallar la probabilidad de que el agente sea consciente de estar experimentando una emoción, hay que invocar al método

obtenerProbabilidadDeEstarConsciente(Emocion) de la clase *DinamicaEmocion*. Se ha utilizado el mecanismo mencionado en el documento [35] para calcular la probabilidad. El funcionamiento es el siguiente: se obtienen los puntos PAD definidos para cada una de las emociones básicas, y sobre cada uno de ellos, se calcula la probabilidad de estar consciente de la emoción correspondiente con la fórmula:

$$\omega_{pe} = \left(1 - \frac{\delta_{pe} - \Delta_{pe}}{\Phi_{pe} - \Delta_{pe}}\right) \cdot i_{pe} \; con \; \Phi_{pe} > \Delta_{pe} \; \forall_{pe} \in \{pe_1, \ldots, pe_5\}$$

ω_{pe}: probabilidad de darse cuenta de la emoción

δ_{pe}: distancia entre el punto PAD del agente y el punto PAD de una emoción básica determinada

Φ_{pe}: umbral de activación

Δ_{pe}: umbral de saturación

i_{pe}: intensidad base de la emoción

pe_n: emoción básica n

Φ_{pe} y Δ_{pe} son configurables para cada agente y esto define el carácter de cada uno de ellos.

Si la probabilidad hallada es superior a cero, entonces indica que la emoción asociada con el punto PAD se ha activado.

6.6.3 SORPRESA

Como se puede observar en la tabla en la que se definieron los puntos PAD de cada una de las emociones básicas, la intensidad base de la emoción sorpresa es 0. Así, si se aplica la fórmula tal y como está definida, el agente nunca es capaz de percibir esta emoción ya que la probabilidad de que se dé cuenta de esta emoción es siempre 0. Por ello, en el caso particular de esta emoción, se considera que la intensidad base varía según el nivel de excitación del agente.

La intensidad base de la emoción sorpresa se calcula en función de la variación del nivel de excitación. Si el nivel de excitación ha variado desde el último ciclo de ejecución superando el umbral U_m entonces, se activa la intensidad base para la emoción sorpresa y asigna como intensidad base de sorpresa el doble del incremento de excitación; y si en este instante coincide que el punto PAD está en la región de SORPRESA, entonces el agente se sorprende.

$$i_b(sorpresa) = 2 \cdot (A_i - A_{i-1}) \;\; para \; A_i - A_{i-1} > U_m$$

A_i: arousal (excitación) en la iteración i

6.6.4 CÁLCULO DEL NIVEL DE VALENCIA

El componente valencia del agente tiende a mantener su equilibrio en todo momento acercándose siempre al valor cero siguiendo la siguiente función de descenso que va relacionado con el nivel de excitación anterior (A_{i-1}) del agente:

$$P_i = P_{i-1} - V_p \cdot A_{i-1}$$

donde V_p es una constante que define la velocidad de descenso.

Cuando el agente consigue ejecutar con éxito un plan, se halla el nuevo valor de placer con la siguiente fórmula:

$$P_i = P_{i-1} + A_{i-1}$$

La razón de utilizar el nivel de excitación anterior (A_{i-1}) es porque cuanto más alto esté el nivel de excitación más probabilidad hay de que el agente esté concentrado en la ejecución del plan por llevar más tiempo ejecutándolo, y por lo tanto más cerca se encuentra de la meta. Esto implica que su expectativa de éxito es cada vez más alta cuando el nivel de excitación va aumentando, y mayor será la intensidad de la valencia obtenida.

Por la misma razón, se ha utilizado la siguiente fórmula para hallar el placer cuando la ejecución de un plan fracasa:

$$P_i = P_{i-1} - A_{i-1}$$

En este caso se resta la excitación anterior (A_{i-1}) al placer anterior (P_{i-1}) para convertir la expectativa de éxito en un estímulo de valencia negativa. Porque cuanto mayor sea la expectativa de éxito, mayor será el daño psicológico que le produce al fracasar.

6.6.5 CÁLCULO DEL NIVEL DE EXCITACIÓN

Para calcular el nivel de excitación (el componente *arousal* del espacio PAD) se extraen los vectores distancia de las entidades que ve el agente con respecto a él; después se calcula la retentiva de cada elemento y se multiplica cada vector distancia con la retentiva de cada elemento. Después se suman todos estos vectores y se compara con el vector que se obtuvo de la misma forma en el ciclo de cálculo anterior. La excitación del agente va en función de la variación de este vector. Cuanta más variación haya, más subirá el nivel de excitación; y cuando no hay variación, el incremento del nivel de excitación será cero. Y estas son las fórmulas utilizadas:

$$A_i = A_{i-1} + \left|\vec{S}_i - \vec{S}_{i-1}\right| \cdot 0.1$$

$$\vec{S}_i = \sum_{m=0}^{n} \vec{s}_m$$

$$Para\ todos\ los\ objetos\ a\ la\ vista\ O_m; \vec{s}_m = \vec{d}_m \cdot \left(1 - e^{\frac{-t}{|\vec{d}_m|}}\right) \cdot \frac{A_c}{|\vec{d}_m|}$$

$$donde\ \vec{d}_m = \overrightarrow{Posicion}(Agente) - \overrightarrow{Posicion}(O_m)$$

A_c: aceleración del organismo= $0.5 \cdot A_i$

$$A_0 = 0.8$$

* 0.1 es un valor para ajustar la variación del estímulo. Para el entorno utilizado, con 0.1 se obtiene un comportamiento razonable de los agentes.

Si los estímulos que ve el agente no varía, su nivel de excitación irá bajando hasta llegar a cero, siguiendo la función de descenso definida en la siguiente fórmula que al igual que en el caso del placer, va en función del nivel de excitación anterior del agente:

$$A_i = A_{i-1} - V_a \cdot A_{i-1}$$

donde V_a es una constante que define la velocidad de descenso.

Para simplificar, se omiten los estímulos táctiles. Pero si el agente pudiese percibir otros tipos de estímulos como el tacto, el sonido, el sabor, el olor o la temperatura, entonces pensamos que la forma de calcular el nivel de excitación sería también **en función de la variación de los estímulos entre un ciclo y el anterior**. Prestando atención primero a los estímulos que varían con más intensidad.

Para el cálculo de la velocidad que puede alcanzar el agente en sus desplazamientos se ha utilizado la siguiente fórmula considerando que tiene una correlación con su nivel de excitación:

$$V_i = A_i \cdot V_{max}$$

donde V_{max} es la velocidad máxima que puede alcanzar el agente. Cuando el nivel de excitación es máximo, su velocidad también será máxima.

6.6.6 CÁLCULO DEL NIVEL DE DOMINIO

Cada vez que el agente consigue terminar con éxito un paso del plan, su dominio se pone en 1 porque siente que está dominando la situación:

$$D_i = 1.0$$

Pero, por el contrario, si la acción falla su dominio se pondrá en -1 porque siente que ha perdido el control de la situación:

$$D_i = -1.0$$

6.6.7 TIPOS DE MEMORIAS UTILIZADAS

Para que el agente evolucione aprendiendo los estímulos del entorno, ha sido necesario implementar una serie de regiones de memoria, cada una especializada en cumplir una función determinada. Se ha implementado un total de cinco regiones distintas de memoria que son:

- **Memoria Implícita**: procesa los estímulos visuales en "crudo", sin intervención de la consciencia, y calcula la intensidad del nivel de alerta que se debe notificar a la parte consciente del agente.

- **Memoria Temporal**: almacena el tiempo medio de ejecución de las acciones y en el caso de que el tiempo de ejecución de una acción es superior al tiempo medio, la concentración del agente disminuye reduciendo la excitación en el espacio PAD. Y en el caso contrario, aumenta la concentración como se vio en los experimentos de Emil Kraepelin. Estas son las fórmulas que se han utilizado para calcular la variación de la excitación según el tiempo de ejecución de una acción:

  Si la memoria temporal,
  - tiene sensación de retraso:

$$A_i = A_{i-1} - A_c \cdot T_f$$

  - no tiene sensación de retraso:

$$A_i = A_{i-1} + A_c \cdot T_c$$

T_c: *constante que representa la tasa de concentración*

T_f: *cosntante que representa la tasa de fatiga*

A_c: *aceleración del organismo*= $0.5 \cdot A_i$

- **Memoria de Trabajo**: es la parte consciente de la mente, necesaria para pensar y para simular los desplazamientos del agente con la mente antes de ponerlos en práctica en el entorno. Es como usar la función ejecutiva del lóbulo frontal del cerebro en las personas.

- **Memoria Procedimental**: se utiliza para ejecutar acciones que el cuerpo lo tiene aprendido (de forma innata o ya asimilada en el crecimiento) sin, o al menos una mínima, intervención de la parte consciente de la mente. Estas acciones aprendidas y conceptualizadas para nuestro agente son: sujetar un objeto, soltar el objeto sujetado,

caminar, levantarse, agacharse, avanzar, levantarse con objeto, y las expresiones corporales de cada emoción. En [25] se habla de la memoria procedural que sirve para almacenar una habilidad y creemos que es equivalente a la memoria procedimental que se ha implementado.

- **Memoria Emocional**: se encarga de almacenar los estímulos emocionales (las valencias) por contexto. El contexto depende del plan, el paso en ejecución, y el estado de ánimo. El estado de ánimo puede ser POSITIVO o NEGATIVO en función de la valencia que tenga el organismo en el momento de ejecutar el paso. Si la valencia es < 0, el estado de ánimo es NEGATIVO y en el caso contrario se considera que es POSITIVO, incluyendo el cero. Participa de forma no consciente en la toma de decisiones. Se dice que los hechos en los que afecta una emoción es reforzada en el hipocampo y perdura en la memoria de forma más intensa. En la implementación, cada vez que el agente se da cuenta de que ha percibido un estímulo emocional, se asocia la valencia del estímulo con el contexto del agente; y perdura como parte de la memoria emocional del agente. De este modo, a partir de la siguiente vez que el agente se encuentre en el mismo contexto, emerge de su parte no consciente la valencia almacenada en la memoria emocional, y el agente lo percibirá como un estímulo fisiológico (interno). Esto hace que un determinado contexto afecte de forma positiva o negativa sin motivo aparente para un observador externo. Pensamos que contribuye en la formalización de la personalidad del individuo mediante la formación de preferencias y rechazos. Esta es la Fórmula de cálculo utilizada para almacenar las experiencias emocionales (*Exp(emo, ctxto)*) de cada contexto en el que emerge alguna emoción:

$$Si\ P_i < 0; Exp_i(emo, ctxto) = min(P_i(emo, ctxto), P_{recuerdo}(ctxto))$$

$$Si\ P_i > 0;\ Exp_i(emo, ctxto) = max(P_i(emo, ctxto), P_{recuerdo}(ctxto))$$

$$Exp_0(emo, ctxto) = 0$$

donde $P_{recuerdo}(ctxto)$ es:

$$\boldsymbol{Si\ |P_i(emo, ctxto)| > |P_{recuerdo\ i-1}(ctxto)|\ \Rightarrow}$$

$$r(ctxto) = 0;$$

$$P_{recuerdo}(ctxto) = P_i(emo, ctxto)$$

$\boldsymbol{Si\ no} \Rightarrow$

$$P_{recuerdo}(ctxto) = P_{recuerdo\ i-1}(ctxto) \cdot e^{\frac{-t}{r(ctxto)}};$$

$$incrementar\ 1\ el\ valor\ de\ r(ctxto)$$

$P_i(emo, ctxto)$: *placer percibido en el momento de sentir una emoción*

$P_{recuerdo}(ctxto)$: *placer recordado para un determinado contexto*

$r(ctxto)$: *nº de veces que se ha recordado algún estímulo relacionado con un contexto*

En la memoria emocional, también se almacenan recuerdos asociados con los objetos presentes en la visión del agente en el momento de percibir un estímulo, que hace emerger una emoción como recuerdos subliminares. Y esta es la fórmula utilizada para almacenar las experiencias emocionales de los objetos vistos $Exp_i(emo, ctxto, entidad)$:

$\boldsymbol{Si\ |Exp_{i-1}(emo, ctxto, entidad)| < |P_i(emo, ctxto)|} \Rightarrow$

$r(ctxto, entidad) = 0;$

$Exp_i(emo, ctxto, entidad) = P_i(emo, ctxto)$

$\boldsymbol{Si\ no} \Rightarrow$

$$Exp_i(emo, ctxto, entidad) = Exp_{i-1}(emo, ctxto, entidad) \cdot e^{\frac{-t}{r(ctxto, entidad)}};$$

$incrementar\ 1\ el\ valor\ de\ r(ctxto, entidad)$

con $Exp_0(emo, ctxto, entidad) = 0$

Tanto $P_{recuerdo}(ctxto)$ como $Exp_i(emo, ctxto, entidad)$ se incluyen finalmente, sumándolos en el valor de la valencia actual del agente para aplicar la influencia de los estímulos recordados por el subconsciente.

6.7 REPRESENTACIÓN DE LOS CONTEXTOS

Los contextos que se guardan en la memoria se representan como una cadena de caracteres por su facilidad de manejo a la hora de generar. Ya que sólo hay que concatenar los distintos estados de las acciones y los comportamientos. Por ejemplo, el contexto representado por la siguiente cadena de caracteres:

"[Coger levantado caja A]"

Esta representación indica que el agente está ejecutando la acción *Coger* y dentro de ella, acaba de ponerse de pie con la *caja A* en sus manos. En el caso de que la acción esté compuesta por otras acciones o comportamientos, se van concatenando los estados internos de cada una de ellas. La siguiente cadena de caracteres es un ejemplo de ello:

"[Transportar [AlcanzarObjetivo [SeguirPared caminar]] caja J al punto [(-12, 0.0, 32.0)]]"

En este caso, indica que el agente se encuentra ejecutando la acción *Transportar* para llevar la *caja J* al punto (-12, 0.0, 32.0), e internamente está ejecutando el comportamiento *AlcanzarObjetivo* y a su vez se encuentra ejecutando el comportamiento *SeguirPared* y dentro de él se encuentra ejecutando el método caminar.

Como se puede observar en los dos ejemplos anteriores, esta representación permite representar el contexto con el nivel de detalle que uno desee. Es importante que cada contexto tenga una representación única, sin coincidir con la representación de otro contexto diferente. Porque de no ser así, los "recuerdos" guardados en la memoria estarían asociados a varios contextos cuando el agente sólo puede encontrarse en un único contexto en un determinado instante de tiempo.

Otra razón por la que se ha escogido esta representación es por su facilidad de interpretar los estados de cada agente a la hora de depurar la aplicación, y ayuda a detectar los errores rápidamente disminuyendo el coste de desarrollo.

6.8 CÁLCULO DE INTENSIDAD DE LOS RECUERDOS

En algunas de las fórmulas presentadas hasta ahora aparece un componente con funciones exponenciales. Estas funciones sirven para calcular la intensidad de los estímulos almacenados en las distintas regiones de memoria.

Según la “curva del olvido” de Ebbinghaus [59] [60], la intensidad de un estímulo disminuye con el tiempo, pero cada vez que se recuerda un estímulo, la velocidad con la que

se olvida es más lenta y perdura más tiempo en la memoria. La siguiente fórmula es una aproximación del comportamiento observado en su experimento [61]:

$$R = e^{\frac{-t}{S}}$$

R: retentiva

t: tiempo transcurrido desde la última vez que se recordó el estímulo (en horas)

S: intensidad relativa del recuerdo

Si observamos las fórmulas utilizadas para almacenar los estímulos en la memoria emocional, vemos que se ha utilizado una versión modificada de esta fórmula:

$$R = e^{\frac{-t}{\#\{r\}}}$$

donde se cambia la variable *S* por *#{r}* que es:

$\#\{r\}$: número de veces que se ha recordado el estímulo

Para ello, el agente tiene memorizado este valor para cada contexto de tal forma que permite calcular la retentiva de cada contexto. Al ir incrementando el número de repetición, la curva del olvido tiene cada vez menos pendiente y el estímulo memorizado persiste durante más tiempo.

La razón de utilizar la fórmula modificada es porque, si se utiliza la valencia recordada como la intensidad relativa del recuerdo, al ir disminuyendo su recuerdo la retentiva obtenida va disminuyendo rápidamente porque la valencia se sitúa entre 0,0 y 1,0. Y esto no es el comportamiento esperado de la curva del olvido. Por eso se ha sustituido por el número de veces que se ha recordado el estímulo. Así en el denominador siempre hay un número mayor o igual a 1, que hace que la retentiva disminuya a una velocidad mucho más lenta.

6.9 MANUAL DEL USUARIO

Una vez creado el mundo de cajas, en esta sección se detallan los prerrequisitos y la forma de manejar el simulador.

6.9.1 PRERREQUISITOS

Para poder ejecutar el simulador, es necesario tener un ordenador con la versión 8 de Java instalada, con el ejecutable java configurado en el PATH del sistema operativo. Para saber cómo realizar esta configuración se puede consultar la referencia [62].

También es necesario tener arrancado el sistema operativo con entorno gráfico y al menos 1024MB de memoria RAM para la máquina virtual de Java. En principio, la librería utilizada soporta los siguientes sistemas operativos: Windows, Mac OS X, Linux/BSD y Solaris. Sin embargo, sólo se ha probado en Windows y Linux.

Aunque no sea necesario, es recomendable tener instalado en el ordenador un chip gráfico con soporte a OpenGL para no saturar el procesador y los procesos del sistema operativo, y obtener un rendimiento óptimo. Es importante tener instalado el controlador de la tarjeta gráfica para OpenGL en el sistema operativo para que se pueda aprovechar su potencia. De no ser así, el simulador no podrá utilizar la opción de aceleración por hardware.

El simulador ha sido probado en los siguientes entornos sin ningún problema:

Configuración Entorno 1:

- Sistema operativo: Windows 10
- Procesador: Intel® Core™ 2 Duo P8600 2.5GHz
- Memoria RAM: 4GB
- Chip gráfico: ATI Mobility RADEON™ HD 4670
- Versión de Java: 1.8.0_66

Configuración Entorno 2:

- Sistema operativo: Ubuntu Desktop 12.02
- Procesador: Intel® Core™ 2 Duo P8600 2.5GHz
- Memoria RAM: 4GB
- Chip gráfico: ATI Mobility RADEON™ HD 4670
- Versión de Java: 1.8.0_66

Configuración Entorno 3:

- Sistema operativo: Windows 10
- Procesador: Intel® Core™ i7-4770K 3.5GHz
- Memoria RAM: 32GB
- Chip gráfico: Intel® HD Graphics 4600
- Versión de Java: 1.8.0_66

6.9.2 CONFIGURACIÓN Y EJECUCIÓN

En esta sección se describe, paso por paso, cómo podemos ejecutar el simulador y finalizarlo posteriormente.

1. El ejecutable está situado en el directorio ***bin*** del artefacto entregado con este documento. Dentro de este directorio hay un script llamado: ***mundoCajas.sh*** para Linux, y un script

llamado: ***mundoCajas.cmd*** para Windows. Una vez configurado el sistema operativo como se indica en la sección 6.9.1, lanzamos uno de estos dos scripts, dependiendo del sistema operativo en el que nos encontremos (es necesario ejecutar el script desde el mismo directorio en el que se encuentra el script, si lo lanzamos desde la línea de comandos). Al lanzar el script se abrirá la ventana de la Ilustración 40. En esta ventana hay 3 secciones. La primera es "Configuración de pantalla" y permite seleccionar la resolución del entorno 3D a simular y a continuación un *checkbox* para activar o desactivar el modo de pantalla completa. La segunda sección: "*Renderer*" es para seleccionar cómo queremos renderizar los fotogramas. Si seleccionamos la opción "*Software Renderer*", todos los cálculos se realizarán en la CPU. Si seleccionamos "*OpenGL Renderer*", todos os cálculos se realizarán en el hardware especializado. Y la última sección titulada "Modo de muestreo" permite elegir el nivel de suavizado de los bordes. Esta opción sólo está disponible cuando se activa la opción *OpenGL Renderer*. El nivel de suavizado crece de izquierda a derecha y a su vez, los recursos del chip gráfico.

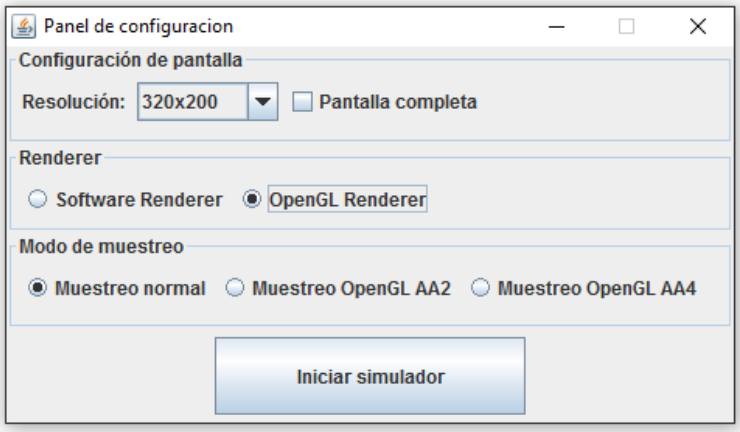

Ilustración 40: ventana de configuración de los gráficos del simulador

2. Una vez configuradas las opciones gráficas en el paso anterior, pulsamos el botón "Iniciar simulador" para arrancar el simulador. Al arrancar, se cerrará la ventana de configuración y se abrirá una ventana que muestra el mundo virtual (Ilustración 41).

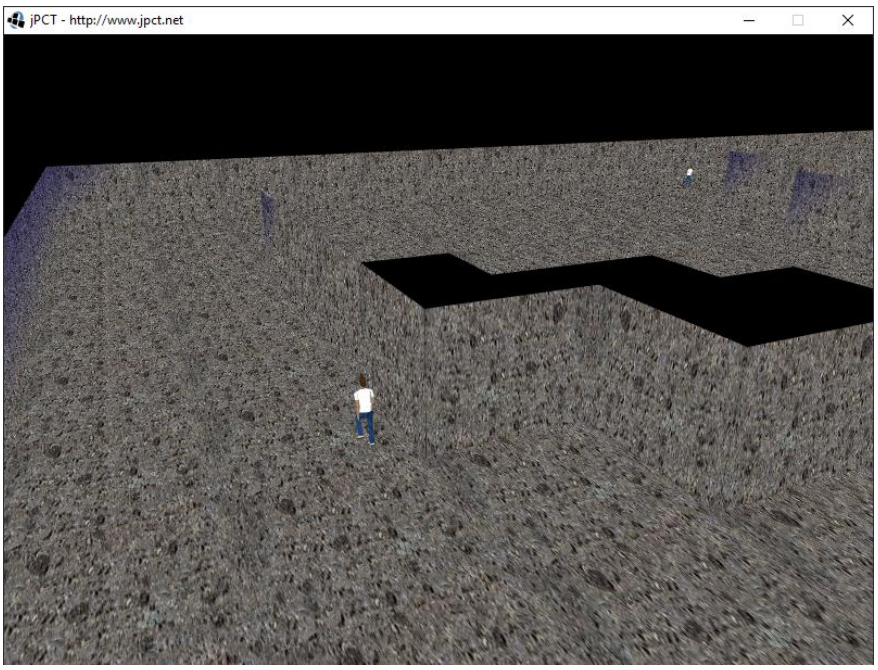

Ilustración 41: ventana que muestra el mundo virtual

3. Para finalizar la simulación, podemos simplemente cerrar la ventana (si no hemos habilitado la opción de pantalla completa), o pulsando la tecla "*Esc*" en el teclado.

6.9.3 NAVEGACIÓN EN EL MUNDO VIRTUAL

En esta sección se detallan las acciones que el usuario puede realizar dentro del entorno virtual.

La cámara principal del simulador la puede mover el usuario. Esto se puede realizar utilizando las teclas de flechas (←, ↑, →, ↓) del teclado o moviendo el ratón. Las teclas de flechas sirven para mover la posición de la cámara dentro del entorno. Estos son los comportamientos de estas teclas:

- Tecla ←: desplaza la cámara hacia su izquierda sin cambiar su orientación.
- Tecla →: desplaza la cámara hacia su derecha sin cambiar su orientación.
- Tecla ↑: desplaza la cámara hacia arriba sin cambiar su orientación.
- Tecla ↓: desplaza la cámara hacia abajo sin cambiar su orientación.

Y con el ratón es posible cambiar la orientación de la cámara:

- Mover el ratón hacia la derecha: gira la cámara hacia la derecha.
- Mover el ratón hacia la izquierda: gira la cámara hacia la izquierda.
- Mover el ratón hacia arriba: gira la cámara hacia arriba.
- Mover el ratón hacia abajo: gira la cámara hacia abajo.

En el caso de que el ratón tenga rueda de desplazamiento, se pueden realizar los siguientes movimientos también:

- Girar la rueda hacia arriba: avanza la cámara en la dirección de su frente.
- Girar la rueda hacia abajo: retrocede la cámara en la dirección de su frente.

Estas opciones sólo estarán habilitadas cuando la cámara principal esté activada. Puesto que en el mundo puede haber varias cámaras. El resto de las cámaras son las que están colocadas como visión de cada uno de los agentes. Por lo que en el mundo habrá tantas cámaras como el número de agentes haya más 1, de la cámara principal. Para conmutar entre todas las cámaras del mundo, basta con pulsar la **tecla C** en el teclado. Con cada pulsación se irá conmutando entre todas ellas y siempre en el mismo orden. Por defecto estará habilitada la cámara principal.

6.9.4 TECLAS DE DEPURACIÓN

Con el propósito de inspeccionar las mallas o el estado interno de cada agente, se han habilitado unas teclas especiales:

- **Tecla B**: muestra u oculta los volúmenes delimitadores (Ilustración 42)

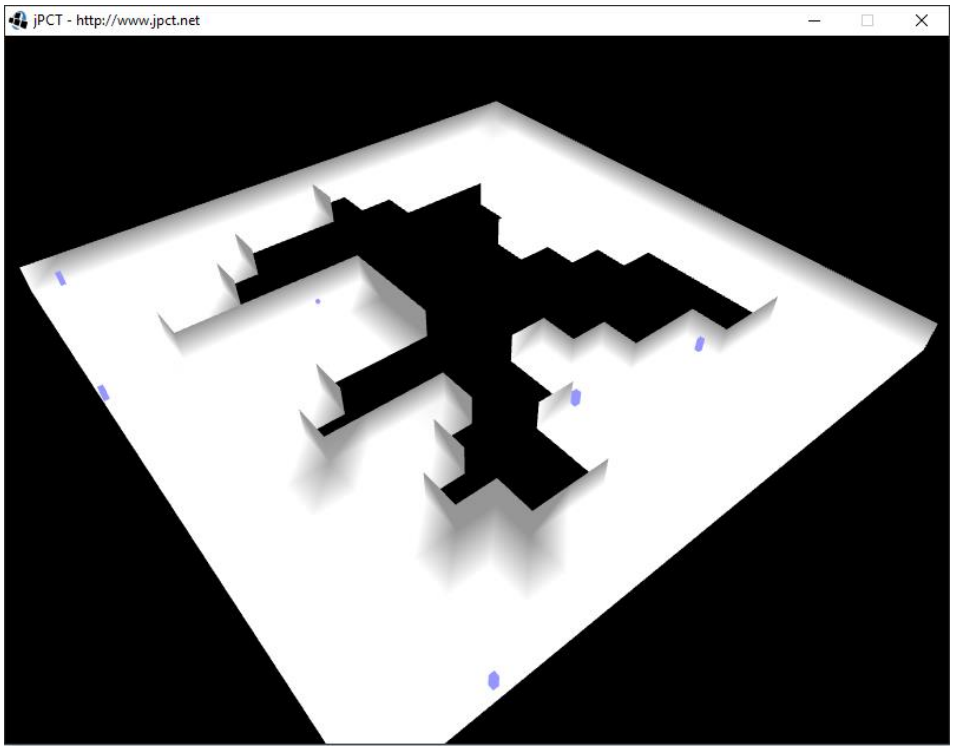

Ilustración 42: pantalla con los volúmenes delimitadores dibujados

- **Tecla W**: muestra u oculta las aristas de los polígonos para encontrar defectos en las mallas. Con la primera pulsación se resaltan las aristas sobre el mundo existente

(Ilustración 43). Y con la segunda pulsación se muestran sólo las aristas eliminando las superficies (Ilustración 44).

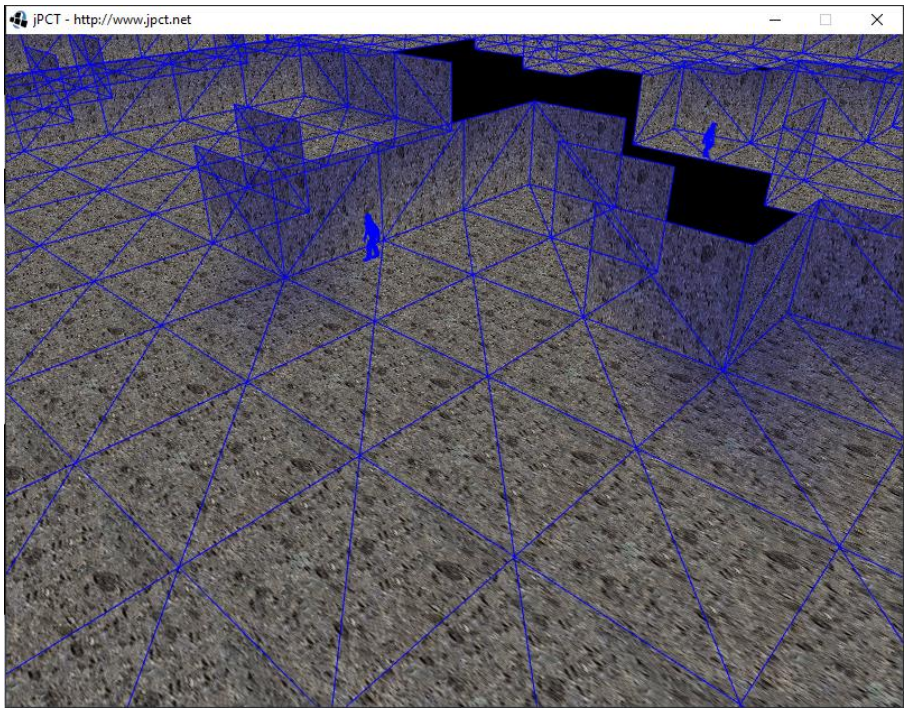

Ilustración 43: la pantalla después de la primera pulsación de la tecla W

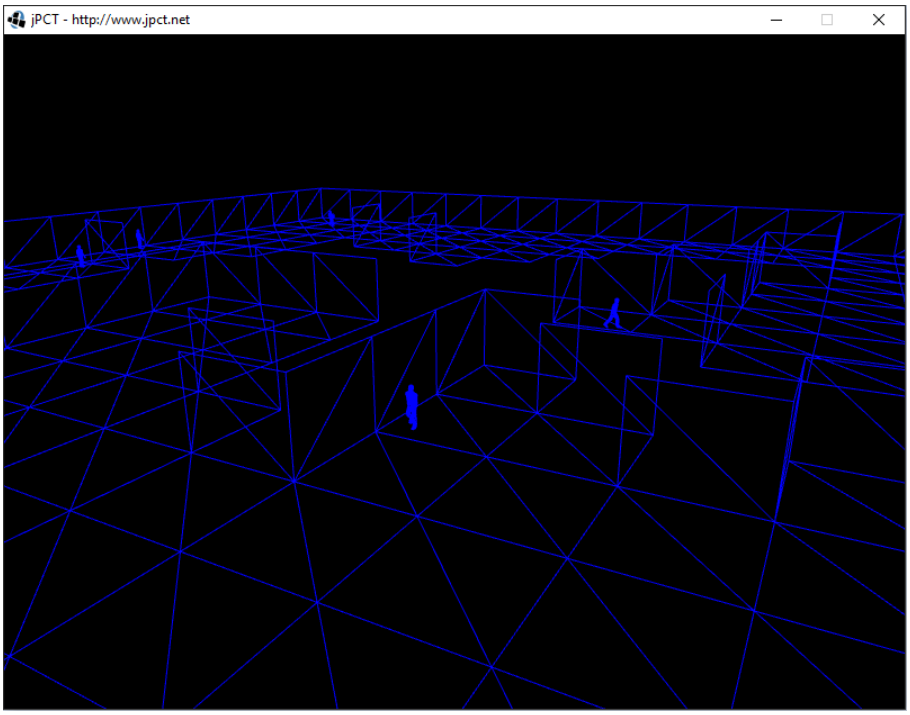

Ilustración 44: la pantalla después de la segunda pulsación de la tecla W

- **Tecla Z**: muestra u oculta el estado interno del simulador y los agentes. Cuando se pulsa esta tecla por primera vez, se muestra en pantalla el tiempo transcurrido desde el inicio de la simulación, los FPS, los UPS, los nombres de los objetos y los puntos

destinos de desplazamiento de los agentes, si está ejecutando algún plan. Cuando algún agente está experimentando una emoción cambia de color el globo que hay encima de sus cabezas. En un estado neutro, el color es azul claro, cuando está alegre: amarillo, cuando está sorprendido: verde, cuando está triste: morado, cuando tiene miedo: gris, y cuando está enfadado: rojo (Ilustración 45). También es posible mostrar el estado interno de cada uno de los agentes. Para ello hay que conmutar la cámara para colocarnos en la vista de algún agente. Una vez hecho esto, pulsamos la tecla Z y se mostrarán las variables internas del agente. Primero se muestra el nombre del agente, después la emoción que está sintiendo (si la hay). A continuación. los valores P (Valencia), A (Excitación) y D (dominio). Después se muestra el nivel de la atención calculada, la posición actual del agente, el deseo a cumplir, los hechos y la acción que está ejecutando. Más abajo se muestra el número de ciclos (de ejecución) en los que se han sentido cada una de las emociones, los deseos a cumplir, los deseos ya cumplidos. Y, por último, recuerdos almacenados en la memoria emocional (Ilustración 46).

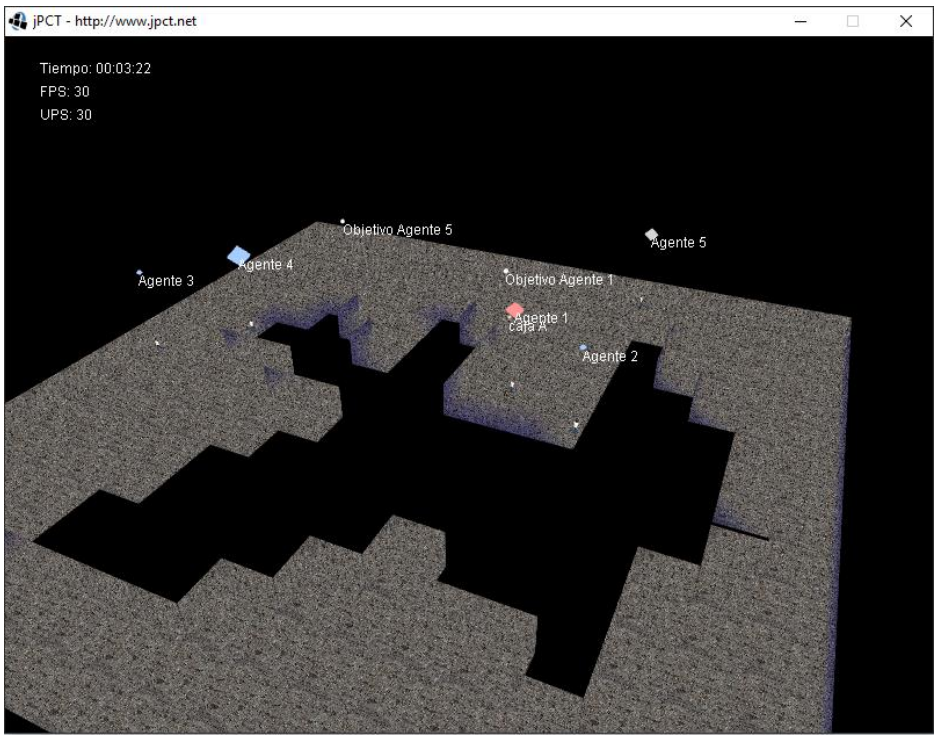

Ilustración 45: pantalla tras pulsar la tecla Z en la cámara principal

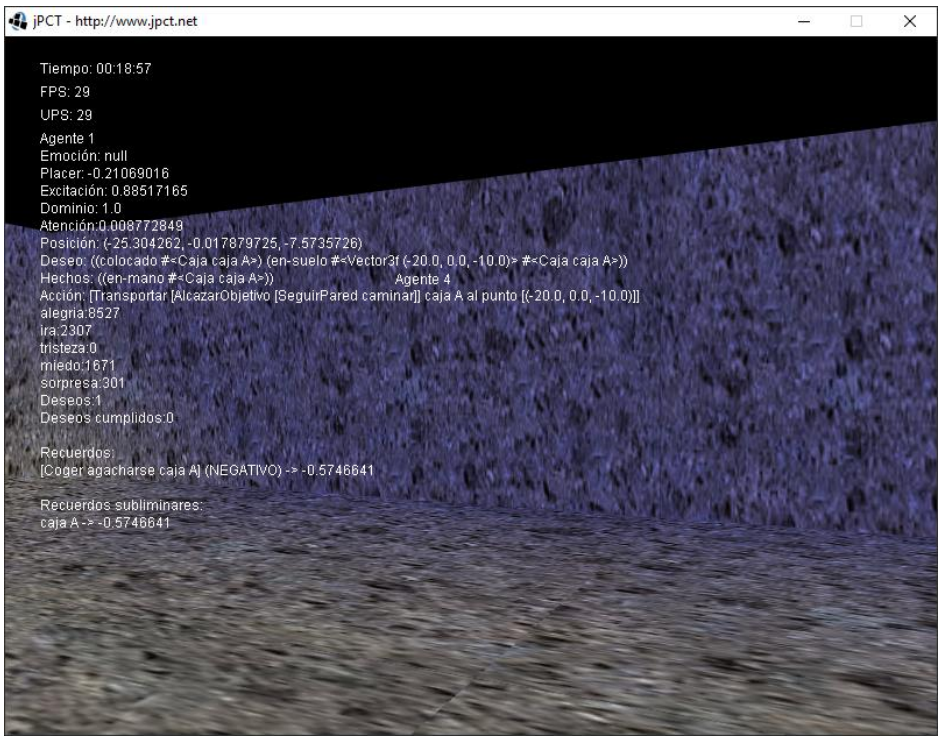

Ilustración 46: pantalla tras pulsar la tecla Z en la vista de un agente

- **Tecla D y P**: en ocasiones, cuando estamos desarrollando, nos encontramos con que el agente está realizando un movimiento que no esperábamos. En estas ocasiones, puede venir bien poder avanzar el ciclo de ejecución del simulador paso a paso para poder parar la imagen en un momento concreto. Para ello, se ha implementado una funcionalidad que está asociada con las teclas D y P. Al pulsar la tecla D la imagen en pantalla se quedará congelada. Esto es porque se queda en suspensión la ejecución del motor físico. En esta situación estarán bloqueadas todas las entidades menos la cámara principal. Al estar la cámara principal desbloqueada, podemos mover la cámara hacia el lugar deseado. Si pulsamos la tecla P en esta situación, podemos avanzar un ciclo de ejecución del simulador, y así podemos ver paso a paso los movimientos de los objetos. Podemos pulsar la tecla P, tantas veces como queramos. Si queremos salir de este contexto volvemos a pulsar la tecla D, y todos los objetos empezarán a moverse de nuevo.

6.10 MANUAL DE REFERENCIA

En esta sección se detallan las acciones que se deben realizar para implementar una nueva aplicación utilizando el simulador 3D.

6.10.1 ENTORNO DE DESARROLLO

El entorno de desarrollo deberá tener instalada la JDK de Java versión 8. Se puede utilizar Eclipse como entorno de desarrollo integrado, pero no es condición necesaria para poder seguir con el desarrollo. Todas las librerías (.jar) necesarias, y que deben ser cargadas en el *classpath*, están dentro del directorio ***lib*** del código fuente. Para lanzar el simulador es necesario añadir un parámetro a la máquina virtual Java para que cargue las librerías nativas del sistema operativo. Estas librerías se encuentran en ***lib/jpct/lib/lwjgl-2.9.1/native***. Para indicar la librería correcta, es necesario configurar la siguiente propiedad del sistema: ***java.library.path***, asignando como valor el directorio de las librerías nativas. Por ejemplo, para Windows es necesario añadir el siguiente argumento en la máquina virtual Java:

-Djava.library.path="{directorio base}/lib/jpct/lib/lwjgl-2.9.1/native/windows"

Como se puede observar, es necesario indicar el directorio con el nombre del sistema operativo en el que se desea ejecutar la aplicación.

6.10.2 CREACIÓN DEL MAIN

Para empezar, se va a describir la forma de crear el método main. Si observamos el contenido la clase *Main* del mundo de cajas que se puede localizar dentro del paquete *es.uc3m.simulador3d.aplicacion*, el método *main*(…) está implementado de la siguiente forma:

```
1  public static void main(String[] args) {
2          PanelDeConfiguracion panel = new PanelDeConfiguracion();
3          panel.mostrar();
4
5          MotorSimulacion motor = FactoriaSimuladorJPCT.crear(JBullet.class,\
                                       panel.getModoVideo(),\
                                       new DecodificadorTecladoAplicacion(),\
                                       new DecodificadorRatonAplicacion(), \
                                       new EntornoCajas());
6
7          motor.arrancar();
8  }
```

En la línea 2 se instancia la ventana que permite configurar las opciones gráficas del simulador. En la línea 3 se invoca su método *mostrar()* para que aparezca la ventana de configuración en pantalla tal y como se mostró en la Ilustración 40. Este método se queda

bloqueado hasta que el usuario pulse el botón de “Iniciar simulador”. Cuando el usuario pulsa sobre este botón, se ejecuta la línea 5. En esta línea se invoca el método de la factoría que se encarga de crear el simulador configurado: *FactoriaSimuladorJPCT.crear(…)*. Al invocar este método se pasan los siguientes argumentos. Primero, el modo de vídeo (configuración de resolución, pantalla completa y el modo de muestreo) configurado por el usuario en el panel de configuración. Segundo, el decodificador de eventos del teclado. Tercero, el decodificador de eventos del ratón. Y cuarto, el entorno a pintar en la ventana de simulación. En la línea 7, se arranca el simulador creado y se invoca el bucle principal de simulación, mostrando a su vez la ventana de la Ilustración 41. Como se puede observar, los pasos que se deben seguir para arrancar el simulador son muy cortos y sencillos. Para crear un nuevo entorno, el desarrollador deberá implementar las clases de los argumentos 2, 3 y 4 de la llamada a la factoría. En las siguientes secciones se detallan qué son y cómo se deben crear estas clases.

6.10.3 DECODIFICADOR DE TECLADO Y RATÓN

Cuando se desea crear un nuevo entorno, el desarrollador puede implementar los decodificadores que necesite adaptándolo a sus necesidades. Pero también puede reutilizar otros decodificadores que se hayan creado para otros entornos si, por ejemplo, las teclas para enviar los eventos al simulador son los mismos.

Si se desea implementar unos nuevos decodificadores, uno de teclado y otro de ratón, es necesario que las clases nuevas implementen las interfaces *DecodifcadorTeclado* y *DecodificadorRaton* (ver Ilustración 11), respectivamente. Y codificar el contenido del método *decodificar(…)* de cada una de ellas. Su implementación es bastante intuitiva. Este método recibe un evento del dispositivo de entrada que encapsula toda la información relativa al evento. Se puede partir utilizando como plantillas las clases ya implementadas: *DecodificadorTecladoAplicacion* y *DecodificarRatonAplicacion*, que se encuentran en el paquete *es.uc3m.simulador3d.aplicacion.entrada.implementacion*.

6.10.4 IMPLEMENTACIÓN DEL ENTORNO

En esta sección se explica cómo se debe crear un nuevo entorno. El entorno contiene el mundo virtual que se puede observar en la ventana de simulación, por lo que suele ser la parte más costosa de elaborar e importante de la aplicación.

Para crear un nuevo entorno, el desarrollador debe crear una clase que implemente de la interfaz *Entorno* (ver Ilustración 11) que se encuentra en el paquete *es.uc3m.simulador3d.motor*. Y es necesario que tenga una referencia a una implementación

de la interfaz *Mundo* (del paquete *es.uc3m.simulador3d.motor*) ya que esta entidad es la que aglutina todos los modelos del entorno virtual y su lógica, y no el *Entorno*. El *Entorno* sólo sirve de interfaz entre el usuario y el mundo virtual. En el método *Entorno.inicializar()* se debe inicializar la instancia de *MundoConcreto* (del paquete *es.uc3m.simulador3d.motor*) registrando todos los modelos que deben aparecer en el mundo virtual. Al instanciar el *MundoConcreto* es necesario pasar como parámetros una implementación del *MundoVisual* y *MundoFisico*, el primero del paquete *es.uc3m.simulador3d.grafico*, y el segundo del paquete *es.uc3m.simulador3d.fisico*. En este caso, como ya hay una implementación para jPCT y JBullet sólo hay que instanciar las clases *es.uc3m.simulador3d.grafico.renderizador.jpct.MundoJPCT* y *es.uc3m.simulador3d.fisico.jbullet.MundoFisicoJBullet*. Una vez instanciada la clase *MundoConcreto,* ya se pueden registrar todos los modelos del sistema. Todos los modelos deben implementar la interfaz *ModeloSistema* del paquete *es.uc3m.simulador3d.motor*. Para registrar el modelo en el mundo hay que invocar al método *ModeloSistema. registrarEnElMundo(mundo)* pasando como argumento el mundo en el que se desea registrar el modelo. Una vez terminada la inicialización, es necesario implementar el método actualizar en el que al menos, se debe invocar al método *Mundo.actualizar(tiempoTranscurrido)* para que avance la simulación. El argumento que hay que pasar es el tiempo transcurrido (en segundos, con decimales) desde el último ciclo de ejecución, que ya viene calculado por el motor de simulación y lo recibe el *Entorno*. El resto de los métodos que hacen falta implementar son para recibir los eventos de los dispositivos de entrada y gestionar el mundo virtual de acuerdo a estos eventos. Para más detalle, se puede ver un ejemplo de implementación mirando el código fuente de la clase *EntornoCajas* que se encuentra en el paquete *es.uc3m.simulador3d.aplicacion.modelo*.

6.10.5 IMPLEMENTACIÓN DEL MODELO DEL SISTEMA

Para implementar un modelo para registrar en el mundo virtual, es necesario crear una clase que implemente de la interfaz *ModeloSistema* del paquete *es.uc3m.simulador3d.motor*. En la aplicación existen ya algunas clases implementadas que pueden servir como base para la creación de nuevos modelos. En el diagrama de clases de la Ilustración 47 se muestra la jerarquía de clases de los distintos tipos de *ModeloSistema* que están creados en la aplicación. La clase abstracta *ModeloSistemaAbstracto* del paquete *es.uc3m.simulador3d.motor* lleva implementado todos los comportamientos que tienen todos los modelos del simulador. Por lo que se recomienda que todos los modelos nuevos que se creen herede de esta clase para no tener que volver a implementar todo, a menos que

la implementación actual no se adapte a las necesidades del entorno a crear. En la aplicación del mundo de cajas hay 2 clases que heredan directamente de esta clase, y son: *es.uc3m.simulador3d.aplicacion.modelo.Campo* que contiene la información de la malla del suelo y las paredes, y *es.uc3m.simulador3d.aplicacion.modelo.Caja* que contiene la malla de la caja que se pinta en el simulador. Es aconsejable seguir este modelo de implementación para los objetos estáticos, es decir, para los objetos que no se pueden mover solos. En cambio, si se quiere crear un modelo autónomo como los agentes del mundo de cajas (*es.uc3m.simulador3d.aplicacion.modelo.agente.AgenteCaja*), se recomienda heredar de la clase abstracta *ModeloAutonomoConVisionAbstracto* del paquete *es.uc3m.simulador3d.motor*. Esta clase, además de los métodos implementados en la clase *ModeloSistemaAbstracto* lleva algunas implementaciones de los métodos definidos en la interfaz *ModeloAutonomoConVision*, y sirve para que el modelo posea su propia vista (cámara) y métodos que permita moverse por el mundo. Existen tres métodos fundamentales que se deben implementar para que el modelo pueda ser autónomo. Estos son los tres métodos:

- ***ModeloSistemaAutonomo.estimular(estimulo)***: este método recibe un estímulo producido en el ciclo de ejecución actual. Internamente el modelo debe almacenar temporalmente el estímulo para procesarlo posteriormente cuando termine el ciclo de ejecución.
- ***ModeloSistemaAutonomo.procesarEstimulos()***: este método es invocado por el motor de simulación antes de terminar cada ciclo de ejecución. Se deben procesar todos los estímulos acumulados en este ciclo y aplicar las transformaciones oportunas al modelo.
- ***ModeloSistemaAutonomo.actuar(tiempoTranscurrido)***: este método es invocado por el motor físico en cada iteración. Es en este método donde se debe realizar los desplazamientos y rotaciones del modelo según la acción que desea realizar el modelo por su cuenta. Por ejemplo, si el modelo desea caminar, hay que avanzar una distancia D calculada con la velocidad del modelo y el tiempo transcurrido, y además avanzar la animación de la malla para reproducir el efecto de “vida” en el modelo.

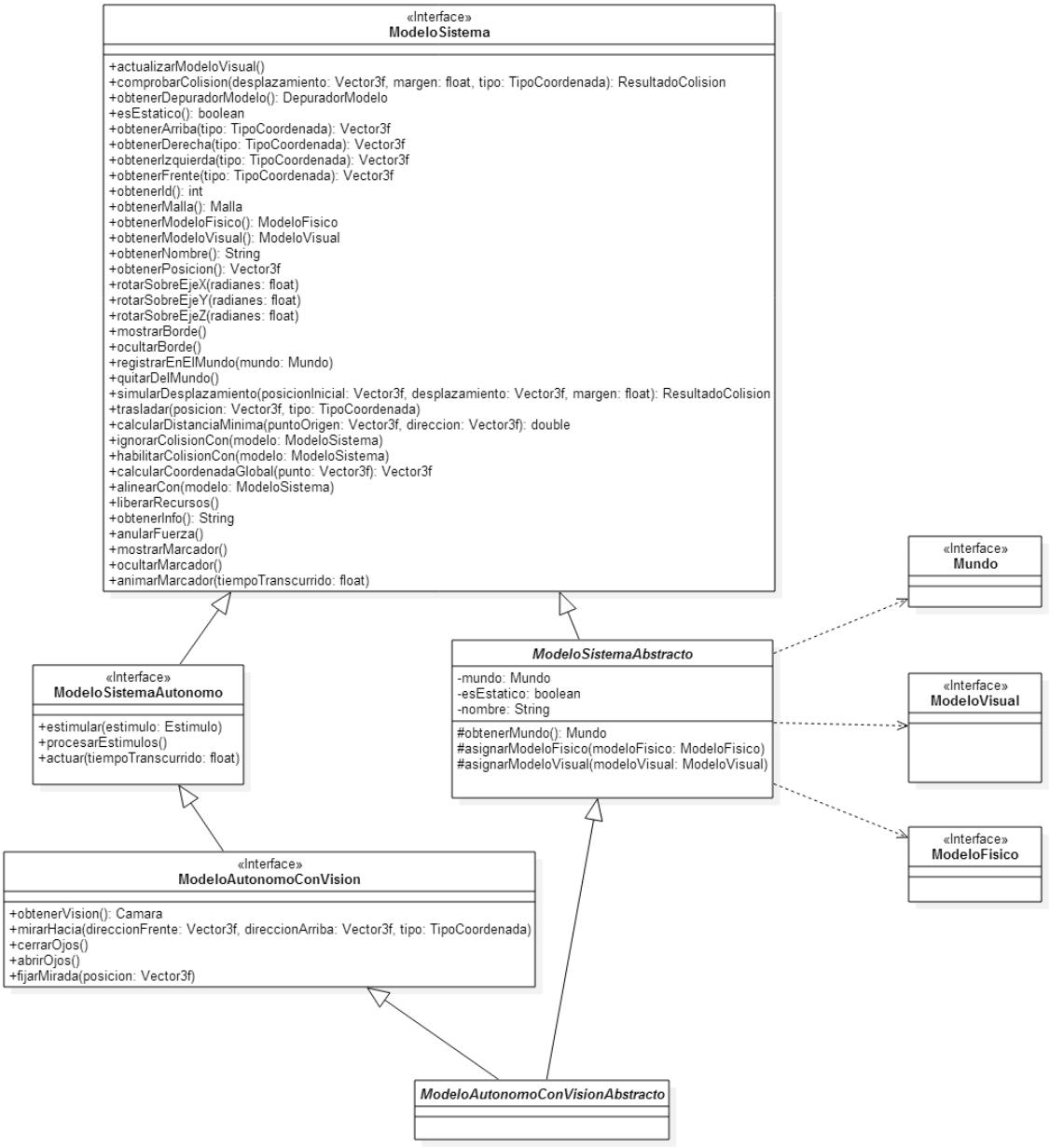

Ilustración 47: jerarquía de clases de *ModeloSistema*

Como se comentó en la sección 5.9, cada modelo del sistema posee su parte visual y su parte física. Se puede ver la relación entre estas clases en el diagrama de clases simplificado de la Ilustración 48. Por ello, toda la implementación de *ModeloSistema* debe inicializar dos modelos visuales y uno físico en su constructor. Uno de los dos modelos visuales es la que contiene la información de la malla que se muestra habitualmente, y el otro modelo visual es el que lleva la información de la malla del *volumen delimitador* que se pintará en pantalla cuando deseemos mostrarlo para algún propósito, por ejemplo, para depurar la fase de detección de colisiones. Como el motor gráfico utilizado es jPCT, es necesario instanciar un objeto del tipo *com.threed.jpct.Object3D* y configurar sus atributos como el nombre de la entidad, la textura, el color, etc. Para conocer más en detalle cómo utilizar la clase Object3D

se puede consultar la documentación de jPCT en su web [44]. Una vez configurados los dos objetos visuales, el primero de ellos, hay que pasar como uno de los parámetros del constructor de la clase *ModeloJPCT* del paquete *es.uc3m.simulador3d.grafico.renderizador.jpct*. El primer argumento del constructor es la instancia de *ModeloSistema* con el que está asociado el modelo visual. El segundo parámetro es la instancia de *Object3D* que se mostrará habitualmente. Y como tercer argumento hay que pasar un valor booleano que indica si crear o no, el globo que se muestra encima de cada modelo como se puede observar en la Ilustración 45. Una vez instanciada la clase ModeloJPCT, para asignar el segundo modelo visual creado, hay que invocar el método *ModeloJPCT.setBorde(borde)* pasando como parámetro el modelo visual con el volumen delimitador. Para crear el modelo físico para JBullet, hay que instanciar un objeto de tipo *ModeloJBullet* del paquete *es.uc3m.simulador3d.fisico.jbullet*. Para ello, hay una clase de utilidad que permite crear el modelo físico con distintos tipos de volúmenes delimitadores. Esta clase se encuentra dentro del paquete *es.uc3m.simulador3d.fisico.jbullet.factoria* y se llama *FactoriaModeloJBullet*. En esta clase existen tres métodos para crear distintos tipos de volúmenes recubridores. Uno permite crear volúmenes en forma de cajas, otro en forma de cápsulas (utilizado en el agente de cajas para minimizar el choque con el suelo), y otro para los objetos estáticos como es el suelo y las paredes. En el último caso, se pasa la malla del objeto visual como uno de los parámetros del método para crear los volúmenes delimitadores que se asemejan al objeto visual. Esto es inevitable ya que se trata del terreno por el que se van a mover los otros objetos. Por eso, la forma de la malla visual y la forma del modelo físico deben ser lo más parecidos posibles. Para más detalle, se puede consultar el *javadoc* de esta clase, y el código implementado en *es.uc3m.simulador3d.aplicacion.modelo.agente.AgenteCaja*.

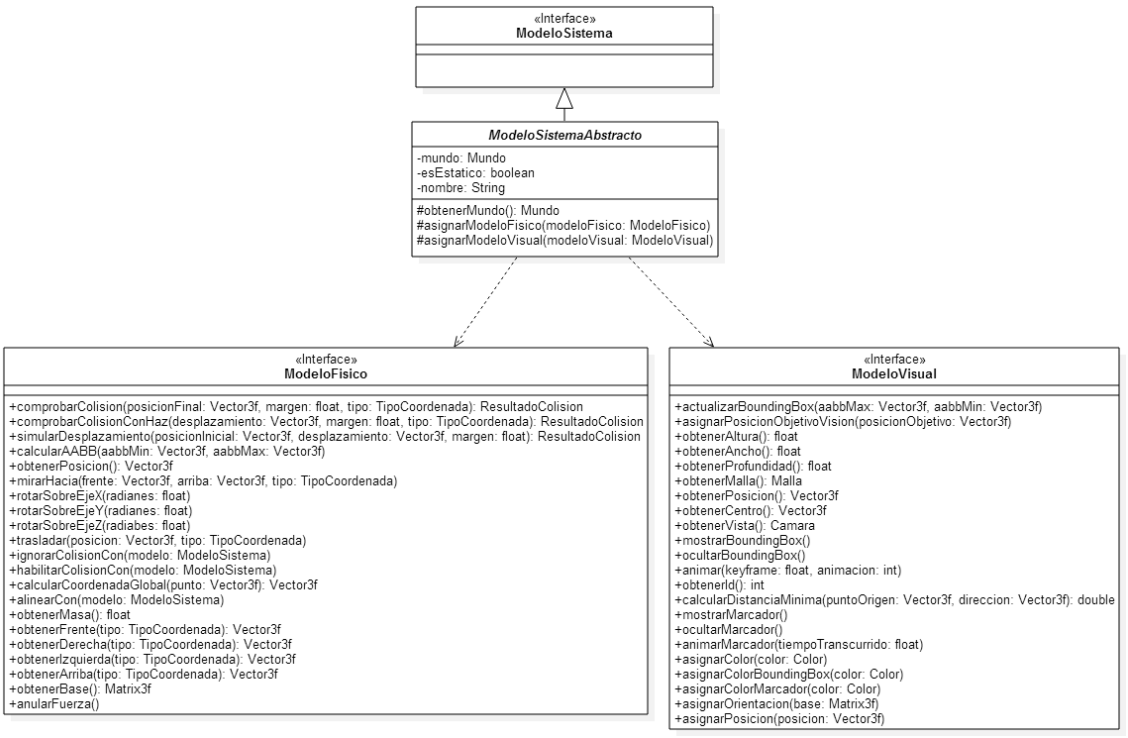

Ilustración 48: relación entre *ModeloFisico* y *ModeloVisual* con *ModeloSistema*

Si no estamos familiarizados con las herramientas matemáticas para realizar los desplazamientos y las transformaciones de los objetos, se recomienda consultar la bibliografía [63] en la que se detallan todos los principios matemáticos fundamentales para trabajar con gráficos por ordenador.

6.10.6 IMPLEMENTACIÓN DE NUEVOS ESTÍMULOS

Actualmente en el simulador existen dos tipos de estímulos: Visual y Fisiológico (ver Ilustración 49). Pero es posible que se desee implementar más tipos de estímulos como el auditivo o el táctil entre otros. Para ello, hay que modificar la interfaz *Estimulo* del paquete *es.uc3m.percepcion* para definir una nueva naturaleza, por ejemplo, la naturaleza auditiva, si se desea generar estímulos auditivos en la simulación. Y a continuación, implementar una clase que implemente de esta interfaz con los métodos y atributos necesarios que permita representar las características del nuevo tipo de estímulo. Después de crear el nuevo tipo de estímulo, hay que generarlos en cada ciclo de actualización del motor físico y enviarlos a cada modelo del sistema. El lugar adecuado para esto es dentro de la llamada al método *actualizar(tiempoTranscurrido)* del *MundoConcreto*. Y después para terminar, hay que implementar los oportunos tratamientos en cada modelo autónomo en el método *ModeloSistemaAutonomo.procesarEstímulos()*.

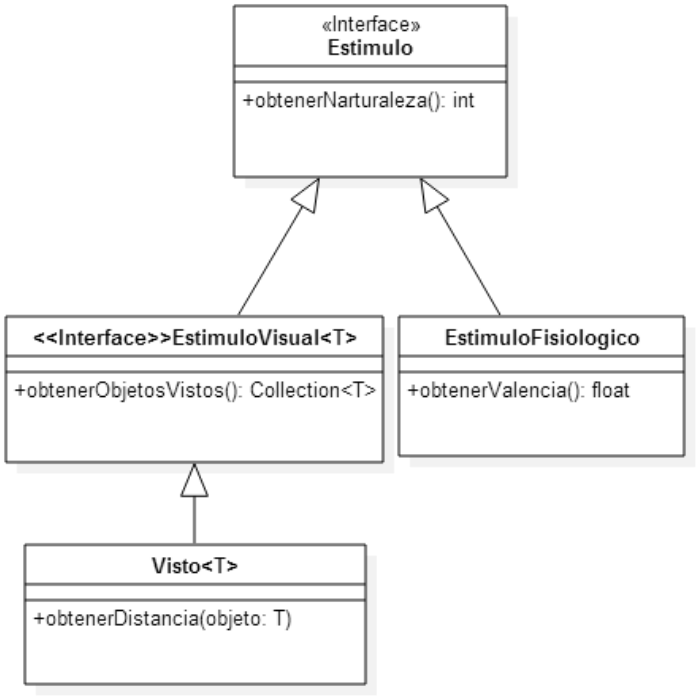

Ilustración 49: diagrama de clases de los estímulos implementados

CAPÍTULO 7: **Resultados**

Una vez explicados todos los detalles de implementación, es el momento de poner en marcha el simulador. Para ello, se ha creado un entorno con cinco agentes, cada uno con una misión distinta, y una sola caja en todo el entorno. Se ha decidido usar esta configuración para observar qué tipo de emociones manifiestan los agentes cuando compitan por la obtención de un único recurso.

A continuación, se detallan todos los parámetros que se han asignado a cada una de las entidades que se han instanciado dentro de este entorno. En la Tabla 2 se detalla las posiciones iniciales de cada entidad, y las velocidades máximas y las misiones de cada agente.

Entidad	Posición inicial	Velocidad máxima	Misión
Agente 1	(-13, 0, 60)	8 unidades/segundo	Colocar Caja A en (-20, 0, -10)
Agente 2	(9, 0, 24)	9 unidades/segundo	Colocar Caja A en (-51, 0, 32)
Agente 3	(34, 0, -18)	7 unidades/segundo	Colocar Caja A en (5, 0, -72)
Agente 4	(-7, 0, -63)	6 unidades/segundo	Colocar Caja A en (-43, 0, -43)
Agente 5	(-30, 0, -15)	10 unidades/segundo	Colocar Caja A en (-30, 0, 36)
Caja	(41, 0, -49)	-	-

Tabla 2: parámetros iniciales de cada entidad 3D

Y en la siguiente tabla (Tabla 3) se detallan los parámetros asignados para cada emoción básica en cada uno de los agentes.

Agente	Emoción	Intensidad base ($\boldsymbol{i_{pe}}$)	Umbral de activación ($\boldsymbol{\Phi_{pe}}$)	Umbral de saturación (Δ_{pe})
Agente 1	Alegría	0.75	0.3	0.21
	Ira	0.75	0.32	0.202
	Miedo	0.75	0.2	0.112
	Tristeza	0.75	0.3	0.223
	Sorpresa	0	0.3	0.252
Agente 2	Alegría	0.75	0.315	0.31
	Ira	0.75	0.319	0.232
	Miedo	0.75	0.315	0.092
	Tristeza	0.75	0.215	0.113
	Sorpresa	0	0.215	0.152
Agente 3	Alegría	0.75	0.31	0.11
	Ira	0.75	0.318	0.132
	Miedo	0.75	0.31	0.212
	Tristeza	0.75	0.31	0.113
	Sorpresa	0	0.31	0.252

Agente 4	Alegría	0.55	0.25	0.11
	Ira	0.55	0.223	0.132
	Miedo	0.55	0.209	0.112
	Tristeza	0.55	0.212	0.193
	Sorpresa	0	0.305	0.112
Agente 5	Alegría	0.35	0.2325	0.21
	Ira	0.35	0.223	0.132
	Miedo	0.35	0.221	0.1212
	Tristeza	0.35	0.221	0.043
	Sorpresa	0	0.323	0.312

Tabla 3: parámetros de emociones para cada agente

La siguiente ilustración muestra el estado inicial del entorno con la configuración descrita (Ilustración 50).

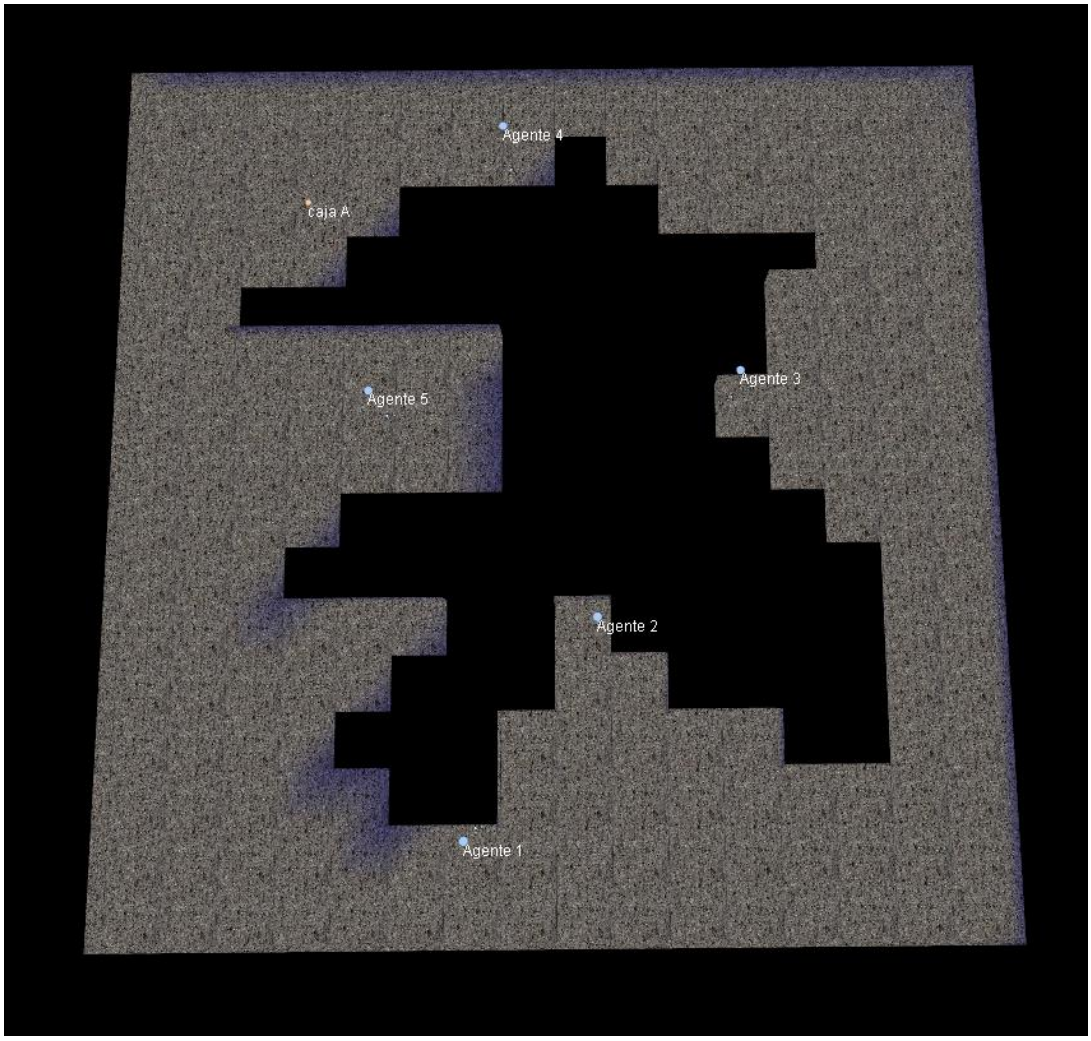

Ilustración 50: estado inicial de la simulación

Durante la simulación se han podido observar las cinco emociones implementadas en distintos agentes.

La emoción ***sorpresa***, cuando se ha encontrado de frente con otro agente en una esquina. Una variación brusca del estímulo visual ha provocado que la intensidad base de la emoción sorpresa varíe de cero (porque la intensidad base de la emoción sorpresa es cero al principio) a un número positivo. Y al coincidir que el punto PAD del agente que estaba dentro de la región de la emoción sorpresa, se ha activado esta emoción. Esta emoción se ha observado más al principio de la simulación y casi nunca después de varias horas de ejecución. Esto se debe a que al principio la memoria del agente apenas está llena de estímulos, y esto hace que cuando se encuentra con otro agente por primera vez, la fórmula de cálculo de excitación arroja un valor que supera el umbral de sorpresa. Sin embargo, cuando se encuentra de nuevo con el mismo agente, la excitación es mucho más pequeña porque su memoria ya tiene este estímulo en su recuerdo. Y al recordar este estímulo se aplica la fórmula de "la curva del olvido" de Ebbinghaus que hace que el estímulo recordado recientemente tenga una "frescura" nula o casi nula, por lo que el nivel de excitación no incrementa apenas.

La emoción **alegría** se ha observado en casi todos los agentes después que conseguir colocar la Caja A en la posición objetivo. Pero no ha sido inmediatamente después de conseguir la meta, sino varios segundos después mientras paseaba por el entorno. Esto se debe a que justo después de conseguir la meta, el organismo del agente alcanza la valencia +1.0 y el dominio +1.0, pero como la región más cercana a la emoción alegría está próxima a la valencia +0.8 y los umbrales de activación configurados en cada agente son muy pequeños, hace falta que la valencia baje más. Y esto se alcanza cuando el agente sigue caminando, pues la valencia va bajando poco a poco por la 'homeostasis' del agente y cuando esta se acerca al +0.8 y la excitación está cerca de +0.8, o +0.2 se produce la alegría.

La emoción **ira** se ha podido observar frecuentemente cuando un agente vuelve a colocar la caja que había colocado ya previamente. Esto se debe a que cuando el agente se da cuenta de que la caja ha sido movida del lugar en el que la dejó la última vez, le produce un estímulo negativo y la sensación de que ha perdido el control de la situación (dominio = -1.0). Cuando el agente se encuentra en este estado y vuelve a coger la caja para transportar a su lugar, el valor de dominio pasa de -1.0 a 1.0. Llegado a este punto, si el agente está excitado con valor próximo a 0.8, es muy frecuente ver que se manifiesta la emoción ira, porque el placer sigue siendo negativo hasta que no consiga ejecutar con éxito el plan (colocar la caja en el punto de destino).

La emoción **tristeza** se manifiesta siempre cuando lleva varias horas de ejecución. Esto es porque, para que se manifieste esta emoción, el nivel de excitación tiene que estar próximo al 0.2. Para que la excitación baje hasta este punto, el agente tiene que haber estado activo

durante varias horas por la velocidad de deceleración del parámetro excitación que está configurado en el proceso de homeostasis.

Y la emoción **miedo** a veces se manifiesta momentos antes de manifestarse la ira. Porque el punto PAD del miedo está justo en el lado opuesto de la ira. Sólo con cambiar el parámetro dominio de 1.0 a -1.0 pasa de la ira al miedo. Observando varias simulaciones, es frecuente ver agentes con miedo cuando se dan cuenta de que la caja que había colocado ha desaparecido. Es como manifestar un miedo 'psicológico' por la incertidumbre que le produce la desaparición de la caja, ‘psicológico’ porque en este entorno no hay ningún tipo de amenaza física, al no existir ni atacantes ni depredadores. Como se mencionó antes, después de producirse esta situación, a veces se puede ver que el agente manifiesta la ira, porque encuentra la caja que desapareció y lo vuelve a coger para llevarlo a su destino y el dominio pasa de -1.0 a 1.0. Se dijo a veces, porque el comportamiento que se manifiesta en el agente cuando siente miedo es el de parálisis y esto hace que los parámetros de placer y excitación se vayan alejando del punto PAD del miedo. Pero si el tiempo de parálisis es muy breve (esto se logra poniendo el umbral de activación y saturación del miedo muy próximos), y el agente logra encontrar la caja enseguida.

CAPÍTULO 8: **Conclusiones**

A lo largo de este proyecto, hemos conseguido crear un mundo virtual tridimensional con agentes capaces de producir cinco emociones básicas. Recapitulamos las tareas que hemos realizado para llegar hasta aquí:

- Creación del simulador de entornos en 3D.
- Creación de las mallas 3D: escenario, caja y agente.
- Creación de las animaciones para cada emoción del agente.
- Implementación del mundo de cajas sobre el simulador 3D.
- Implementación del planificador de alto nivel.
- Implementación del planificador de bajo nivel.
- Implementación de las acciones del agente.
- Implementación de los comportamientos auxiliares del agente.
- Implementación de los algoritmos de cálculo del punto PAD y cálculo de emociones.
- Implementación de las distintas regiones de memoria del agente.

Y a continuación detallamos las conclusiones que hemos obtenido tras terminar todas las fases de implementación, y el análisis de resultados del experimento final.

Se ha llegado a varias conclusiones sobre el modelo PAD. El modelo parece tener sentido después de ver la simulación final y viendo manifestarse las emociones básicas y pudiendo dar un explicación lógica y coherente a cada una de ellas.

Se considera que la excitación del organismo es el complementario del cansancio, y correlativo con el concepto de 'energía' de la persona. Aunque los agentes implementados no se alimentan con comida para recuperar su 'energía', para poder volver a moverse una vez que el nivel de excitación llega a cero, debe dormir un tiempo para aumentar el nivel de excitación del organismo. Esto hace que podamos afirmar que el concepto de excitación en el modelo PAD sea sinónimo de 'energía' que puede gastar el agente.

La excitación también es sinónimo de 'aceleración del organismo'. Todos los procesos internos del cuerpo están correlacionados con el nivel de excitación. Cuanto más alto es el nivel de excitación más velocidad puede alcanzar el agente, y el proceso de homeostasis también es más rápido, lo que significa mayor velocidad para obtener el equilibrio mental y ser menos emocional.

En cuanto a la implementación de la memoria emocional basada en las observaciones de LeDoux y Damasio, por desgracia, sus efectos no han sido muy notables. Quizás porque el

entorno no aprovecha su utilidad al no existir ningún tipo de peligro físico para el agente. Al no existir ningún tipo de amenaza, no se generan tantos estímulos negativos e intensos que se queden grabados en sus mentes. Las intensidades de los estímulos memorizados por un agente en el entorno de cajas son muy débiles y apenas afectan en sus comportamientos.

El hecho de tener el simulador 3D ha ayudado mucho a la hora de reflexionar y plasmar los distintos mecanismos del cerebro humano, extraídos de toda la bibliografía consultada de este documento.

CAPÍTULO 9: **Futuras líneas/Trabajos**

En el futuro, sería interesante crear entornos con peligro físico 'real' para ver si la memoria emocional es útil para mejorar la capacidad de supervivencia de los agentes.

También sería interesante añadir la posibilidad de que los agentes puedan procesar más tipos de estímulos como el sonido, el tacto, el olor o la temperatura. Pensamos que así se podría observar mayor variedad de reacciones al poder, el agente, detectar mayor cantidad de señales de peligro. Y esto nos podría ayudar a pensar en un modelo para la toma de decisiones y ver las influencias de los distintos parámetros del organismo.

Otro posible trabajo interesante sería: definir acciones alternativas para que el agente tenga que elegir entre las alternativas, y así observar la influencia de la memoria emocional en el proceso de toma de decisiones.

Como en este trabajo sólo se han implementado cinco tipos de emociones, se podría implementar la emoción básica que falta en la lista de Ekman, que es la de *aversión*. Pero para abordar esta tarea, se considera que es necesario implementar un estímulo que sea capaz de producir una valencia negativa desde el interior del cuerpo. Esto es porque consideramos que la aversión se produce al rechazar el propio cuerpo del agente (fisiológicamente, y no conscientemente) algo del exterior. Por ejemplo, cuando olemos comida podrida, se podría producir esta emoción, con el fin de hacernos alejar de ella, para no ingerirla y así evitar enfermar o morir. Un posible escenario apto para producir esta emoción sería un escenario con comida, colocadas en varios puntos del mundo, y que el agente sea capaz de olerla para distinguir si se encuentra en buen estado o mal estado. En este escenario, el agente puede ir a por alimentos para reponer energías y recibir estímulos positivos. Incluso se podría colocar distintos tipos de alimentos para ver si afloran preferencias hacia un tipo determinado de comida, o no.

Por último, se podría considerar un cambio de arquitectura del simulador a una arquitectura cliente-servidor, situando el motor gráfico en el lado servidor y los agentes como clientes en otro u otros ordenadores, anticipando a la necesidad de más recursos computacionales, con la implementación de mejoras más complejas en los agentes.

CAPÍTULO 10:**Bibliografía y Referencias**

[1] T. Taniguchi, イラストで学ぶ人工知能概論, Tokio: Kodansha, 2014.

[2] T. Odaka, 初めての AI アプリケーション, Tokio: Ohmsha, 2010.

[3] «The Physical Symbol System,» [En línea]. Available: http://www.faculty.umb.edu/gary_zabel/Courses/Bodies,%20Souls,%20and%20Robots/Texts/AI%20and%20Robotics/The%20Physical%20Symbol%20system.htm.

[4] T. Doi, M. Fujita y H. Shimomura, 脳・身体・ロボット 知能の創発をめざして, Tokio: Maruzen Publishing, 2012.

[5] N. Yoshida, やわらかい情報処理, Tokio: Saiensu, 2003.

[6] C. Becker, S. Kopp y I. Wachsmuth, *Why emotions should be integrated into conversational agents,* Bielefeld: Universidad de Bielefeld, 2001.

[7] C. Becker-Asano, *WASABI: Affect Simulation for Agents with Believable Interactivity*, Bielefeld: Universidad de Bielefeld, 2008.

[8] «Second Life Official Site - Virtual Worlds, Avatars, Free 3D Chat,» [En línea]. Available: http://secondlife.com. [Último acceso: 04 03 2015].

[9] «LSL-Editor,» [En línea]. Available: http://wiki.secondlife.com/wiki/LSL-Editor. [Último acceso: 03 04 2015].

[10] «LSL-Plus,» [En línea]. Available: http://wiki.secondlife.com/wiki/LSL-Plus. [Último acceso: 03 04 2015].

[11] «Second Life Viewer,» [En línea]. Available: https://secondlife.com/support/downloads/. [Último acceso: 03 04 2015].

[12] «JGomas, Game Oriented Multiagent System based on Jade,» [En línea]. Available: http://gti-ia.dsic.upv.es/sma/tools/jgomas/index.php. [Último acceso: 03 04 2015].

[13] Ochakko, ３D ゲーム・クックブック, Tokio: Shuwa System, 2007.

[14] «3DS MAX,» [En línea]. Available: http://www.autodesk.es/products/3ds-max/overview. [Último acceso: 03 04 2015].

[15] «MAXScript Introduction,» [En línea]. Available: http://docs.autodesk.com/3DSMAX/14/ENU/MAXScript%20Help%202012. [Último acceso: 03 04 2015].

[16] Z. Nishikawa, 3D graphics techniques to become a Game Developer, Tokio: Impress Japan, 2013.

[17] L. Benstead, D. Astle y K. Hawkins, Beginnin OpenGL Game Programming, Second Edition, Boston, Massachusetts: COURSE TECHNOLOGY, 2009.

[18] «Havok,» [En línea]. Available: http://www.havok.com. [Último acceso: 03 04 2015].

[19] «Bullet Physics Library, Real-Time Physics Simulation,» [En línea]. Available: http://bulletphysics.org/wordpress. [Último acceso: 03 04 2015].

[20] «Open Dynamics Engine,» [En línea]. Available: http://www.ode.org. [Último acceso: 03 04 2015].

[21] «Home of the Blender project - Free and Open 3D Creation Software,» [En línea]. Available: http://www.blender.org. [Último acceso: 03 04 2015].

[22] T. Mullen, Introducing Character Animation With Blender, Indianapolis, Indiana: Wiley Publishing, 2007.

[23] «GIMP - GNU Image Manipulation Program,» [En línea]. Available: https://www.gimp.org/. [Último acceso: 02 01 2016].

[24] «Terragen 3,» [En línea]. Available: http://planetside.co.uk/products/terragen3. [Último acceso: 02 01 2016].

[25] J. LeDoux, The Emotional Brain: The Mysterious Underpinnings of Emotional Life, Nueva York: Simon and Schuster, 1996.

[26] «Plutchik's Wheel of Emotions,» [En línea]. Available: https://en.wikipedia.org/w/index.php?title=Plutchik%27s_Wheel_of_Emotions&redirect=no. [Último acceso: 03 04 2015].

[27] D. Evans, EMOTION: A Very Short Introduction, Oxford: Oxford University Press, 2001.

[28] «Robert Plutchik,» [En línea]. Available: http://es.wikipedia.org/wiki/Robert_Plutchik. [Último acceso: 03 04 2015].

[29] «PAD emotional state model,» [En línea]. Available: http://en.wikipedia.org/wiki/PAD_emotional_state_model. [Último acceso: 03 04 2015].

[30] A. Damasio, The Feeling of What Happens: Body and Emotion in the Making of Consciousness, Nueva York: Harcourt Brace & Company, 1999.

[31] J. LeDoux, Synaptic Self: How Our Brains Become Who We Are, Nueva York: Viking, 2002.

[32] A. Damasio, Looking For Spinoza, Nueva York: Harcourt, 2003.

[33] E. Fox, Rainy Brain, Sunny Brain: How to Retrain Your Brain to Overcome Pessimism and Achieve a More Positive Outlook, Londres: WILLIAM HEINEMANN, 2012.

[34] A. Damasio, DESCARTE'S ERROR, Nueva York: InkWell Management, 1994.

[35] C. Becker-Asano y I. Wachsmuth, «Affective computing with primary and secondary emotions in a virtual human,» *Autonomous Agents and Multi-Agent Systems,* vol. 20, nº 1, pp. 32-49, 2010.

[36] «INKSCAPE,» [En línea]. Available: https://inkscape.org. [Último acceso: 11 01 2016].

[37] «StarUML,» [En línea]. Available: http://staruml.io/. [Último acceso: 11 01 2016].

[38] «Eclipse Luna,» [En línea]. Available: https://www.eclipse.org/luna/. [Último acceso: 11 01 2016].

[39] «Git,» [En línea]. Available: https://git-scm.com/. [Último acceso: 03 01 2016].

[40] «Google Drive,» [En línea]. Available: https://www.google.com/intl/es_es/drive/. [Último acceso: 03 01 2016].

[41] «Google Keep,» [En línea]. Available: https://www.google.com/keep/. [Último acceso: 03 01 2016].

[42] «Java SE - Downloads | Oracle Technology Network | Oracle,» [En línea]. Available: http://www.oracle.com/technetwork/es/java/javase/downloads/index.html.

[43] A. Davison, Killer Game Programming in Java, Sebastopol, California: O'REILLY, 2005.

[44] «jPCT 3D engine: the free 3D solution for Java and Android,» [En línea]. Available: http://www.jpct.net/. [Último acceso: 03 01 2016].

[45] «JBullet - Java port of Bullet Physics Library,» [En línea]. Available: http://jbullet.advel.cz/.

[46] «LWJGL - Lightweight Java Game Library,» [En línea]. Available: https://www.lwjgl.org/. [Último acceso: 03 01 2016].

[47] F. Dunn y I. Parberry, 3D Math Primer for Graphics and Game Development, Plano, Texas: Wordware Publishing, 2008.

[48] «Left- vs. Right-handed coordinate systems,» [En línea]. Available: https://www.evl.uic.edu/ralph/508S98/coordinates.html. [Último acceso: 03 01 2016].

[49] Y. Okawa, 物理エンジン Bullet プログラミング, Tokio: I/O BOOKS, 2010.

[50] «jPCT's coordinate system,» [En línea]. Available: http://www.jpct.net/wiki/index.php?title=Coordinate_system. [Último acceso: 03 01 2016].

[51] T. Hirayama, ゲームプログラマになる前に覚えておきたい技術, Tokio: Shuwa System, 2008.

[52] T. Akenine-Möller, E. Haines y N. Hoffman, Real-time Rendering, Third Edition., Wellesley, Massachusetts: A K Peters, 2008.

[53] C. Ericson, Real-time Collision detection, San Francisco, California: MORGAN KAUFMANN PUBLISHERS, 2005.

[54] Ohuchi, Yamamoto y Kawamura, マルチエージェントシステムの基礎と応用, Tokio: CORONA PUBLISHING, 2007.

[55] «Clojure,» [En línea]. Available: http://clojure.org. [Último acceso: 03 04 2015].

[56] S. Halloway, Programming Clojure, Raleigh, Carolina del Norte: The Pragmatic Bookshelf, 2009.

[57] J. Gregory, Game Engine Architecture, Wellesley, Massachusetts: A K Peters, 2009.

[58] «The Psychology of Fatigue: Work, Effort and Control,» [En línea]. Available: https://books.google.es/books?hl=es&lr=&id=34GkWLvvc_cC&oi=fnd&pg=PR11&dq=The+Psychology+of+Fatigue:+Work,+Effort+and+Control. [Último acceso: 03 04 2015].

[59] «Curve of Forgetting,» [En línea]. Available: https://uwaterloo.ca/counselling-services/curve-forgetting. [Último acceso: 03 04 2015].

[60] «THE CURVE OF FORGETTING,» [En línea]. Available: http://ol.scc.spokane.edu/jroth/Courses/English%2094-study%20skills/MASTER%20DOCS%20and%20TESTS/Curve%20of%20Forgetting.htm. [Último acceso: 03 04 2015].

[61] «Forgetting curve,» [En línea]. Available: https://en.wikipedia.org/wiki/Forgetting_curve. [Último acceso: 03 04 2015].

[62] «How do I set or change the PATH system variable?,» [En línea]. Available: https://www.java.com/en/download/help/path.xml. [Último acceso: 05 01 2016].

[63] J. M. Van Verth y L. M. Bishop, Essential Mathematics for games & interactive applications, Second Edition, Burlington, Massachusetts: MORGAN KAUFMANN PUBLISHERS, 2008.

[64] S. AUTOR, Opengl, Memphis, Tennessee: Books LLC, 2010.

[65] «Clojure,» [En línea]. Available: http://clojure.org/. [Último acceso: 03 01 2016].

[66] «American Scientist - Plutchik,» [En línea]. Available: http://web.archive.org/web/20010716082847/http://americanscientist.org/articles/01articles/Plutchik.html. [Último acceso: 03 04 2015].

ANEXOS

ANEXO 1:Evolución histórica de la tecnología 3D

En enero de 1992, Silicon Graphics Inc. (SGI) lanza el estándar OpenGL (versión 1.0), para permitir el acceso estandarizado a las tarjetas gráficas. Con esto, cada fabricante debía implementar las API de OpenGL si querían que sus dispositivos fuesen compatibles con este estándar. Actualmente cada fabricante proporciona su implementación de OpenGL como parte de su controlador (*driver*). Este año SGI crea el *architectural review board* (OpenGL ARB), que se trata de un grupo de compañías que mantendrán y definirán las nuevas especificaciones de OpenGL. La base de la primera versión de OpenGL era el IrisGL: una API desarrollada por SGI en los años anteriores, y que se había convertido en un estándar de facto en la industria. El principal problema que tenía IrisGL era su dependencia al dispositivo gráfico subyacente. Esto se solucionó en OpenGL simulando por software todas aquellas funcionalidades que no estuviese soportado por el hardware. Permitiendo crear aplicaciones gráficas avanzadas en máquinas que no sean estaciones de trabajo.

Desde 1988 hasta 1993, era común el uso de flat shading: representación de modelos mediante polígonos con colores, pero sin ningún tipo de texturas.

En 1993, se empiezan a utilizar las texturas, permitiendo pegar imágenes sobre las superficies de los polígonos.

En 1994, se lanzan las consolas PlayStation y Sega Saturn. A partir de esto los gráficos 3D en tiempo real evoluciona de forma vertiginosa, superando a la tecnología de gráficos 3D utilizados en los PC de la época.

En 1995, empieza a proliferar la tecnología de gráficos 3D para PC con la aparición de Windows 95. Microsoft proporcionó con él, el DirectX. Empezaron a extenderse las tarjetas gráficas 3D para Windows. Sin embargo, aún no era posible manejar gráficos 3D en tiempo real porque el pipeline aún no tenía la suficiente capacidad.

En 1997, con el lanzamiento de DirectX 5 permitió el manejo de gráficos 3D en tiempo real. Por esta época NVIDIA lanzó la serie RIVA 128 y ATI la serie RAGE 3D. Tres años después del lanzamiento de la PlayStation, se consiguió trabajar con gráficos 3D en tiempo real de similares características en los PC. La industria de la informática se pone las pilas y los gráficos por ordenador empiezan a evolucionar de forma acelerada. En enero de este mismo año, se lanza la versión 1.1 de OpenGL, enfocado a dar soporte a las texturas.

En 1998, se lanza DirectX 6 que implementaba algunas técnicas desarrolladas para generar gráficos 3D en tiempo real, permitiendo el procesado de imágenes a nivel de pixel.

En marzo de este año se lanza OpenGL 1.2 soportado por las tarjetas Rage 128, Rage 128 GL, Rage XL/XC, Rage 128 Pro, Rage Fury MAXX y todas las tarjetas posteriores. En octubre se lanza la versión 1.2.1 de OpenGL con soporte a texturas múltiples (*multi-texture*) y estaba soportado por las tarjetas: Radeon, Radeon Mobility, Radeon 7500 Mobility, Radeon 8500, Radeon 9000, Radeon 9200, Radeon 9600, Radeon 9800, GeForce 3, GeForce 4Ti, GeForce FX y todas las posteriores tarjetas gráficas.

En 1999, se lanza el DirectX 7. Trabaja a nivel de vértice o polígono. Implementaba mecanismos para permitir la transformación de coordenadas por vértices y procesamiento de iluminación mediante hardware. Se denominaban hardware T&L (Transform & Lighting). Las tarjetas compatibles con DirectX 7 son: NVIDIA GeForce 256 y los primeros ATI RADEON. Los gráficos por PC supera la capacidad de las video consolas domésticas.

En 2000, se lanza al mercado la PlayStation 2 con una potencia gráfica similar a los PC más modernos de la época. Se propone un mecanismo para permitir procesar los gráficos mediante programas ejecutados dentro de las Unidades de Procesamiento Gráfico (GPU), y aparece el concepto de *Programmable Shader*. A finales de año Microsoft lanza la versión 8 de DirectX que daba soporte a la arquitectura hardware con *Programmable Shader*. DirectX 8 soportaba dos tipos de *Shaders* programables: *Programmable Vertex Shader* y *Programmable Pixel Shader*. El primero trabaja a nivel de vértice, y el segundo a nivel de pixel. Cada *shader* se ejecuta en una unidad específica dependiendo de su tipo. Los programas s*hader* permiten añadir una funcionalidad gráfica nueva a la GPU por software. Las GPU que tenían soporte DirectX 8 son NVIDIA GeForce3 y ATI RADEON 8500.

En 2001 aparece Windows XP en el cual venía integrado el DirectX 8. A finales de año Microsoft lanza también la primera versión de Xbox basada en DirectX 8. Esta fue la primera video consola doméstica del mundo que utilizaba una arquitectura con *Programmable Shader*. Este mismo año se lanzó OpenGL 1.3.

En 2002 se publica DirectX9 que soporta la versión 2.0 de la especificación de *Programmable Shader*. Se empiezan a sonar las siglas SM (Shader Model) para especificar la generación de la arquitectura del *Programmable Shader*. Las GPU que soportan DirectX9 son compatibles con SM 2.0, siendo 2.0 la versión del *Programmable Shader* que soporta la GPU. Las unidades que soportan SM 2.0 son capaces de ejecutar programas *shader* más largos, poseen mayor número de instrucciones y permiten realizar cálculos de mayor precisión que las anteriores SM 1.x. Este año aparece la versión 1.4 de OpenGL, soportado por las tarjetas: Quadro DCC, Quadro4 380 XGL, Quadro4 500XGL, 550XGL, Quadro4, 700XGL, 750XGL, 900XGL, 980XGL y todas las tarjetas posteriores.

En 2003 se publica OpenGL 1.5 con soporte a *vertex buffer object* (VBO).

En 2004 Microsoft lanza una versión actualizada de DirectX9 con soporte a la especificación SM 3.0 que elimina el límite del tamaño de los programas y aumenta aún más el juego de instrucciones. Aparece una nueva interfaz de bus: "PCI-Express" que aumentaba considerablemente el ancho de banda de las tarjetas gráficas. Este año NVIDIA se adelanta a ATI y lanza la serie 6 de GeForce con arquitectura SM 3.0. Se lanza también OpenGL 2.0 que daba soporte a *vertex shader* y *fragment shader* mediante el lenguaje ensamblador ARB. Estas son las tarjetas gráficas con soporte a OpenGL 2.0: Radeon 9500, 9550, 9600, 9650, 9700, 9800, X1050, Radeon Xpress 200/1100, Radeon X300, GeForce 6800, Quadro 600, Quadro FX 500 y todas las posteriores versiones.

En 2005, ATI lanza, con un año y medio de retraso que NVIDIA, las GPU que soportan SM 3.0: la serie RADEON X1800, aunque no soportaba del todo la especificación 3.0. NVIDIA lanza la segunda generación de GPU con soporte a SM 3.0: la serie GeForce 7800. A finales de año Microsoft lanza la segunda generación de su video consola, la Xbox 360 con soporte a DirectX 9/SM 3.0 con una GPU diseñada por ATI.

En 2006, GeForce lanza la tercera generación de GPU con SM 3.0. Su competencia, ATI, lanza la serie RADEON X1900 aún sin soportar del todo las funcionalidades de SM 3.0. A finales de año SONY lanza su nueva video consola: la PlayStation 3 con GPU de NVIDIA con arquitectura SM 3.0. Se lanza también la versión 2.1 de OpenGL que incluye la revisión 1.20 de su lenguaje de *shader*: GLSL. El OpenGL 2.1 estaba soportado por las tarjetas Radeon HD 2350, GeForce FX y posteriores versiones de estas.

En 2007, Microsoft lanza DirectX 10 con su nuevo sistema operativo Windows Vista. La versión del *shader* programable en DirectX 10 es la 4.0. En esta versión aparece un nuevo tipo de s*hader* llamado "Geometry Shader". Este nuevo *shader* trabaja a nivel de vértice al igual que "Vertex Shader". Su función es la de disminuir o aumentar el número de vértices programáticamente. Las tarjetas con soporte a DirectX 10 son la serie GeForce 8000 de NVIDIA y la serie RADEON HD 2000 de ATI. Estos utilizan una arquitectura de *shader* unificada (Unified Shader Architecture) que incorpora varias unidades de *shader* programables genéricos en vez utilizar una específica en cada fase de procesamiento en el pipeline. Esto permite asignar todas unidades disponibles en las fases que requieran una mayor capacidad de procesamiento.

En 2008, Microsoft publica el Windows Vista Service Pack 1 y con él aparece el DirectX 10.1. El *Shader Model* correspondiente es el 4.1. Las tarjetas con soporte a esta arquitectura fueron sólo de AMD (ATI): la serie RADEON HD 3000. Este mismo año se lanza OpenGL 3.0 con soporte a *framebuffer objects*. Las tarjetas con soporte a OpenGL 3.0 son la serie Radeon HD, GeForce 8 y sus posteriores versiones de tarjetas. La arquitectura hardware

necesaria para aprovechar al máximo las API de OpenGL 3.0 es equivalente a la arquitectura de las tarjetas con soporte a DirectX 10. Esto indica que OpenGL, tras un período de estancamiento desde la versión 2.0, ha logrado remontar hasta llegar al nivel de su principal competidor.

En 2009, NVIDIA lanza GeForce 210 y 220 que son sus primeras GPU con soporte para DirectX 10.1 y SM4.1. A finales de año Microsoft lanza Windows 7 con DirectX 11 con SM 5.0. Esto añade 4 nuevos s*haders* en su arquitectura, 3 programables y 1 fijo. Los nuevos *Shaders* programables son: "Hull Shader", "Domain Shader" y "Compute Shader". El fijo se denomina "Tessellator". En esta nueva arquitectura de *shader* es posible utilizar subrutinas mediante enlaces dinámicas a ellas. Esto mejora la "programabilidad" de los *shaders* ya que permite reutilizar trozos de código ya implementados sin tener que volver a copiarlos. Este año se lanzan dos versiones de OpenGL: 3.1 en marzo, con GLSL 1.40, y OpenGL 3.2, en agosto, con GLSL 1.50.

En 2010, se lanzan otras dos versiones de OpenGL, pero en esta ocasión las dos a la vez: la 3.3 y la 4.0. La primera con GLSL 3.30 y la segunda con GLSL 4.00. La arquitectura hardware óptima para utilizar las API de OpenGL 4.0 es la del DirectX 11 que incluye un "Tessellator" mediante hardware. Por otro lado, el OpenGL 3.3 incluye unas extensiones ARB para permitir emular el "Tessellator" por software y así dar soporte a las funcionalidades de OpenGL 4.0 en los hardware antiguos como los que soportan DirectX 10.1 e inferiores.

Las grandes marcas como AMD, NVIDIA o Intel están trabajando para lograr la fusión entre los CPU y las GPU que permite procesar gráficos 3D mediante software ya que la tecnología actual lo permite sin depender de hardware de aceleración gráfica. Como primeras versiones de estas "CPU/GPU", Intel lanzó en 2010 sus microprocesadores Core i3, Core i5 y Pentium con GPU integrada con soporte para DirectX 10. Y AMD lanzó en 2011 la serie "Fusion APU" con soporte para DirectX 11.

En 2012, Microsoft lanza Windows 8 y con él vino integrado el DirectX 11.1. Como indica la versión, se hicieron cambios menores a su predecesor DirectX 11 y no tuvo ningún cambio significativo de pipeline, ni de número, ni características de unidades de procesamiento por lo que sigue siendo SM 5.0. Una de las mejoras realizadas fue dar soporte a *shaders* de baja precisión. Probablemente porque Windows 8 da soporte a CPU de ARM, muy utilizados en *tablets* y *smartphones*. Así permite dar soporte a un amplio número de GPU de distintas capacidades de procesamiento. Las primeras tarjetas gráficas que se comercializaron con soporte a DirectX 11.1 vino primero de AMD: la RADEON HD 7970 y posteriormente, la GeForce GTX 680 de NVIDIA. Este mismo año se publica la versión

4.3 de OpenGL con GLSL 4.30 que equivale al SM 5.0 de Direct3D. A finales de este año Nintendo lanza su nueva videoconsola: la Wii U con GPU con arquitectura SM 5.0.

En 2013, se libera Windows 8.1 y con él, el DirectX 11.2, que al igual que su predecesor, se realizaron algunas mejoras y no tiene ningún cambio significativo en cuanto a la arquitectura se refiere. Por lo que sigue siendo SM 5.0. A finales de este año aparecen 2 nuevas videoconsolas con soporte a SM 5.0. Primero Sony lanza la PlayStation 4 y posteriormente, Microsoft lanza la Xbox One, las dos con GPU de AMD.

Referencias: [16] [52] [64]

ANEXO 2:Modelado del avatar con Blender

En este apartado detallaremos los pasos que se han seguido para modelar el avatar en Blender.

A2.1 MODELADO DE LA MALLA

Partimos de la pantalla inicial (Imagen 1).

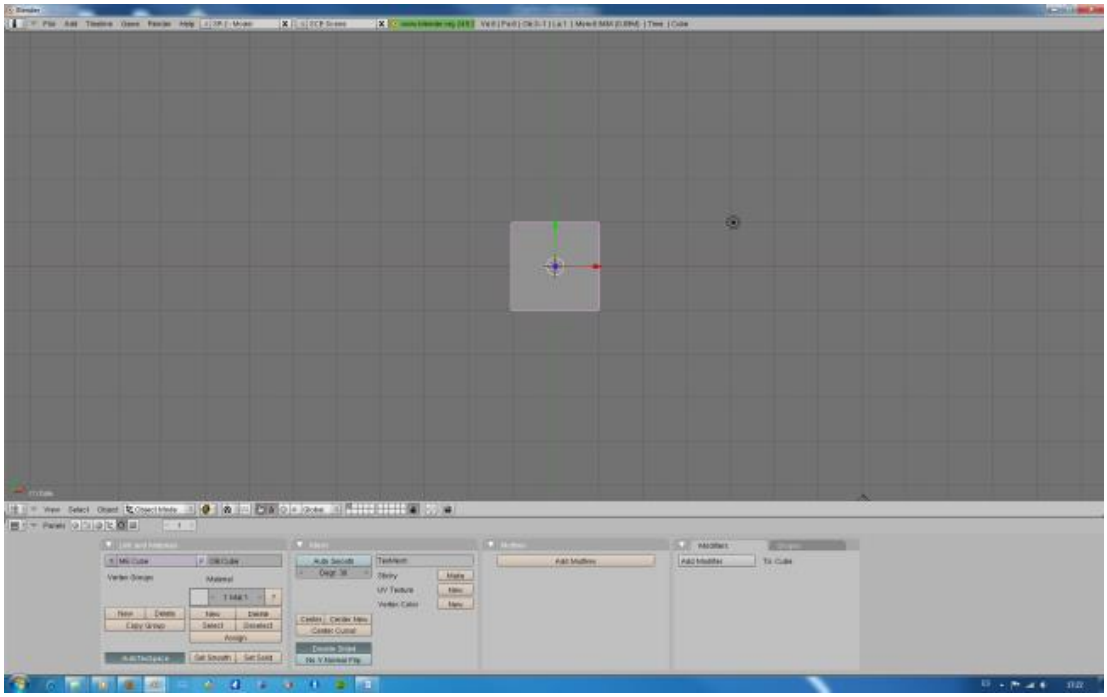

Imagen 1

Para empezar a manipular el cubo pulsamos Tabulador para entrar en modo de edición (Imagen 2).

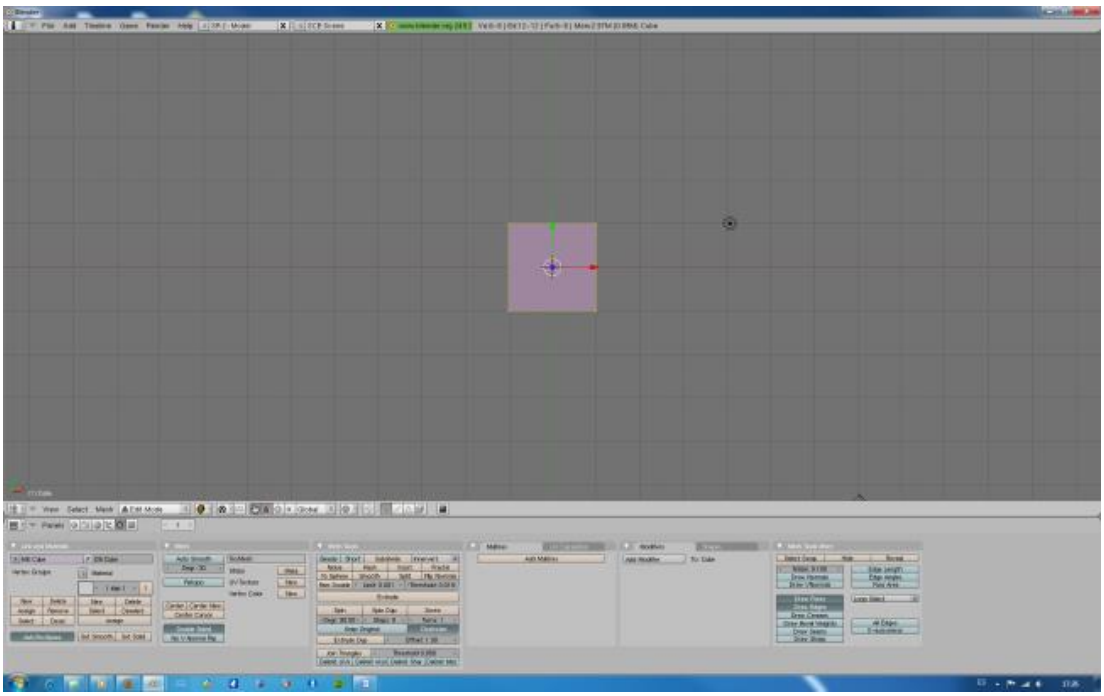

Imagen 2

Ahora pulamos sobre la pestaña 'Modifiers' -> 'Add Modifier' -> 'Mirror' (Imagen 3) para aplicar un efecto espejo sobre el cubo. Esto creará automáticamente un reflejo el modelo sobre el eje Y. De este modo sólo tenemos que modelar la mitad del cuerpo del avatar y finalmente persistir el modelo reflejado.

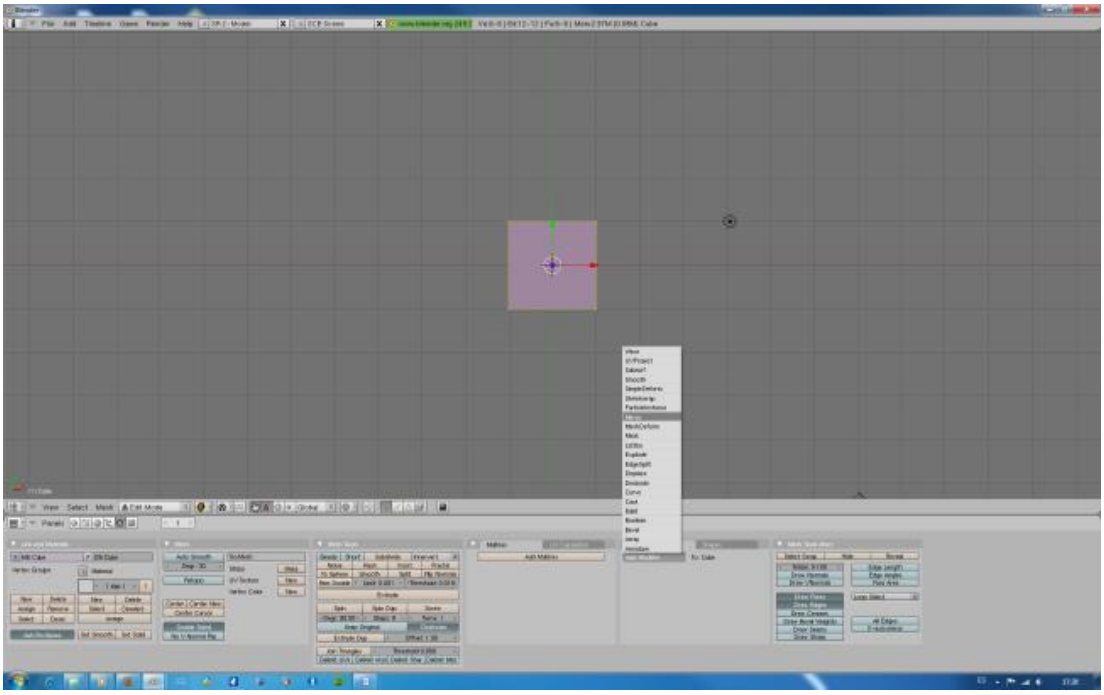

Imagen 3

En el recuadro que nos aparece seleccionamos “X” y “Do Clipping” (Imagen 4), para que nos cree el reflejo sobre el eje Y, y seleccionando “Do Clipping” nos recortará toda parte del modelo que sobresalga del eje Y.

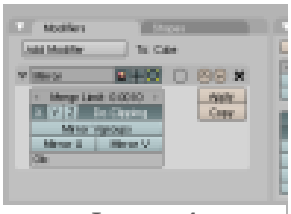

Imagen 4

Pulsamos el botón derecho del ratón sobre el borde inferior del visor 3D y seleccionar “Split area” (Imagen 5) para dividir la ventana.

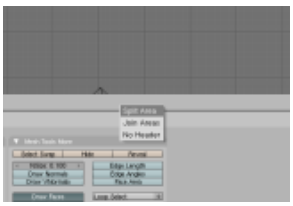

Imagen 5

A continuación, desplazamos el cursor hasta la mitad del visor 3D. Y hacemos clic con el botón izquierdo del ratón para confirmar (Imagen 6).

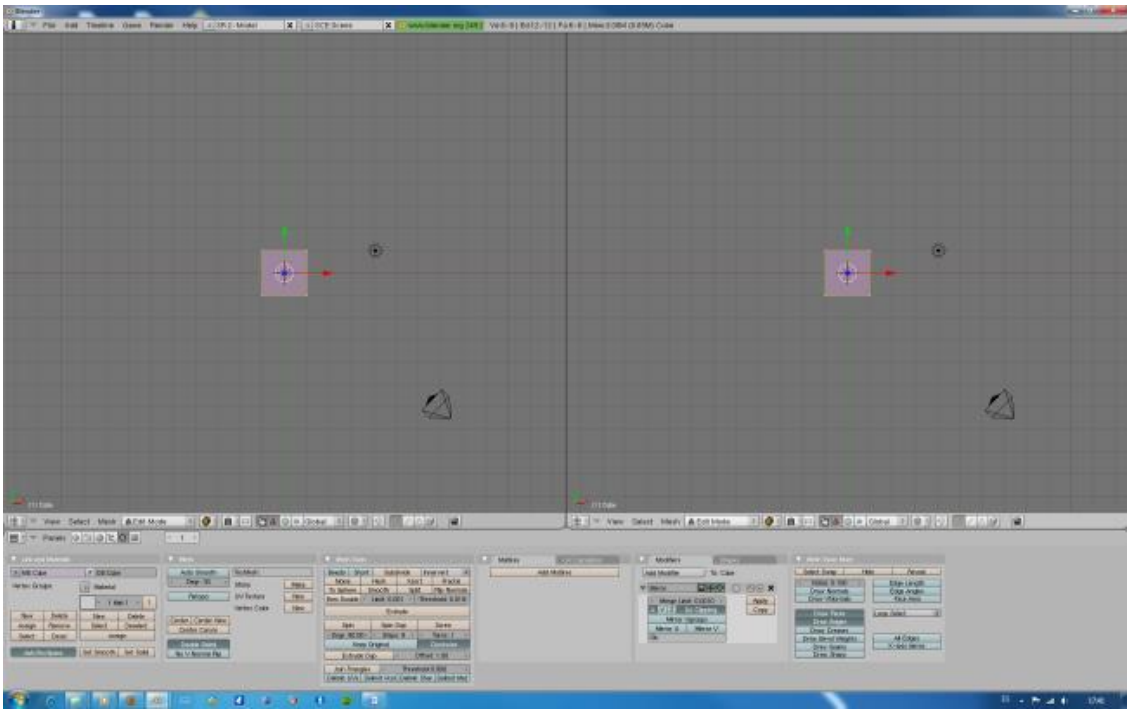

Imagen 6

Seleccionamos la ventana de la izquierda y pulsamos 'View' -> 'Front' (Imagen 7) para visualizar el modelo de frente.

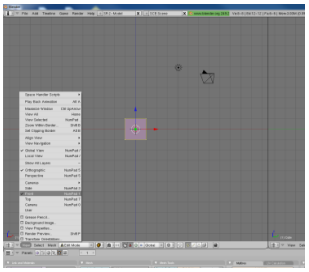

Imagen 7

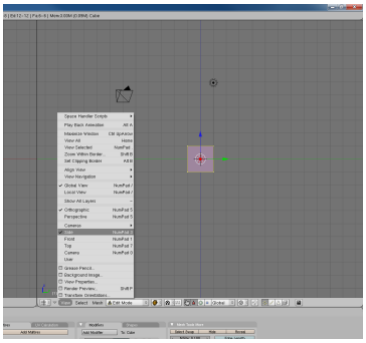

Imagen 8

Y en la ventana de la derecha y pulsamos 'View' -> 'Side' (Imagen 8) para visualizar el modelo de un lado.

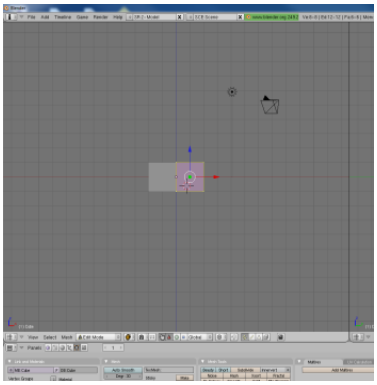

Imagen 9

Desplazamos el cubo hacia la derecha en la vista frontal (Imagen 9). Pulsamos “A” para deseleccionar todo y mantener pulsado “B” hasta que aparezca un círculo gris. Seleccionamos los dos vértices inferiores del cubo introduciéndolos dentro del círculo. Así queda seleccionada la cara inferior del cubo (Imagen 10).

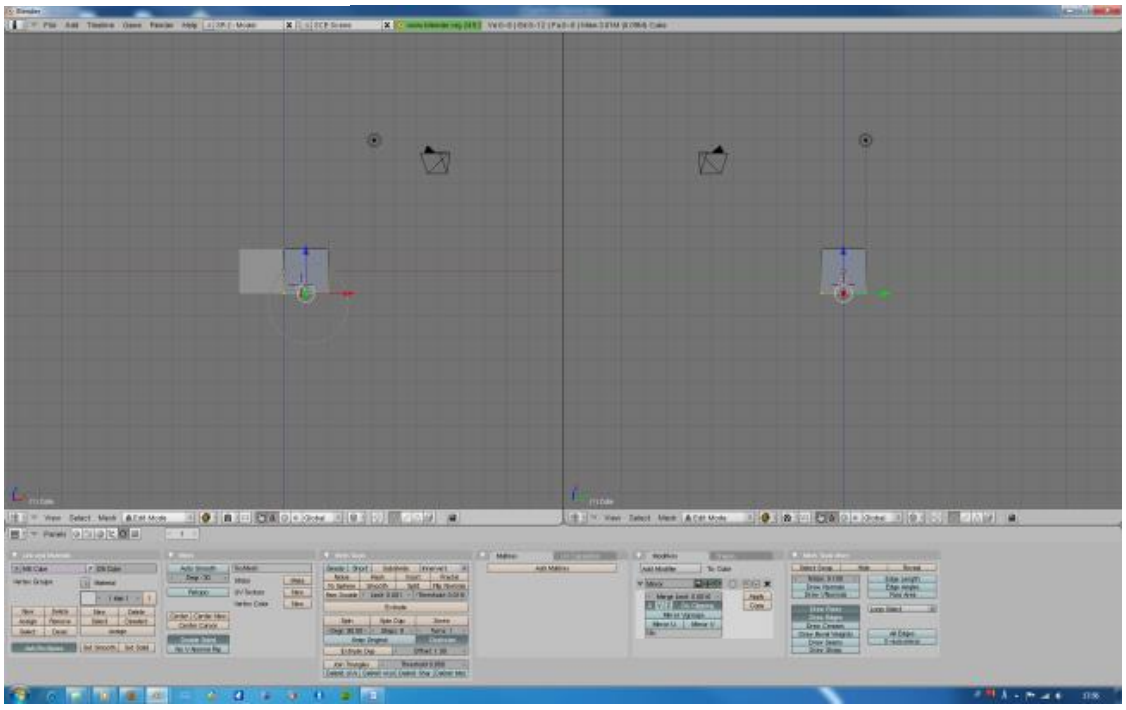

Imagen 10

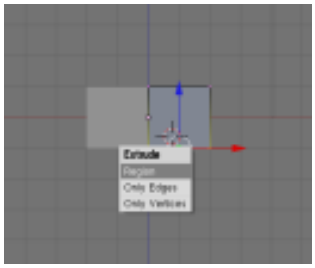

Imagen 11

Pulsamos “E” en el teclado y elegimos “Region” para alargar el cubo (Imagen 11).

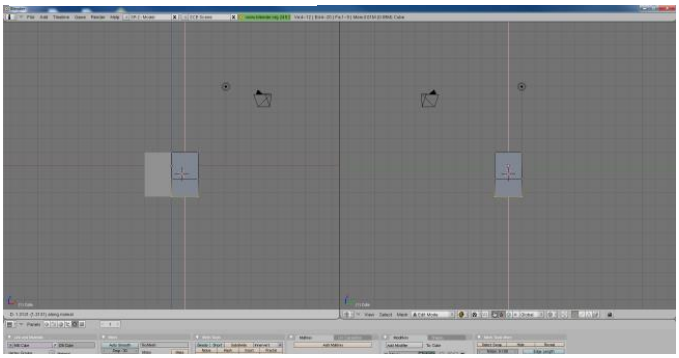

Imagen 12

Alargamos el cubo hacia abajo y hacemos clic izquierdo con el ratón para terminar la operación (Imagen 12).

Ahora vamos a crear una pierna. Para ello vamos a extrudir la cara inferior del cubo hacia abajo pulsando la tecla “E” (Imagen 13).

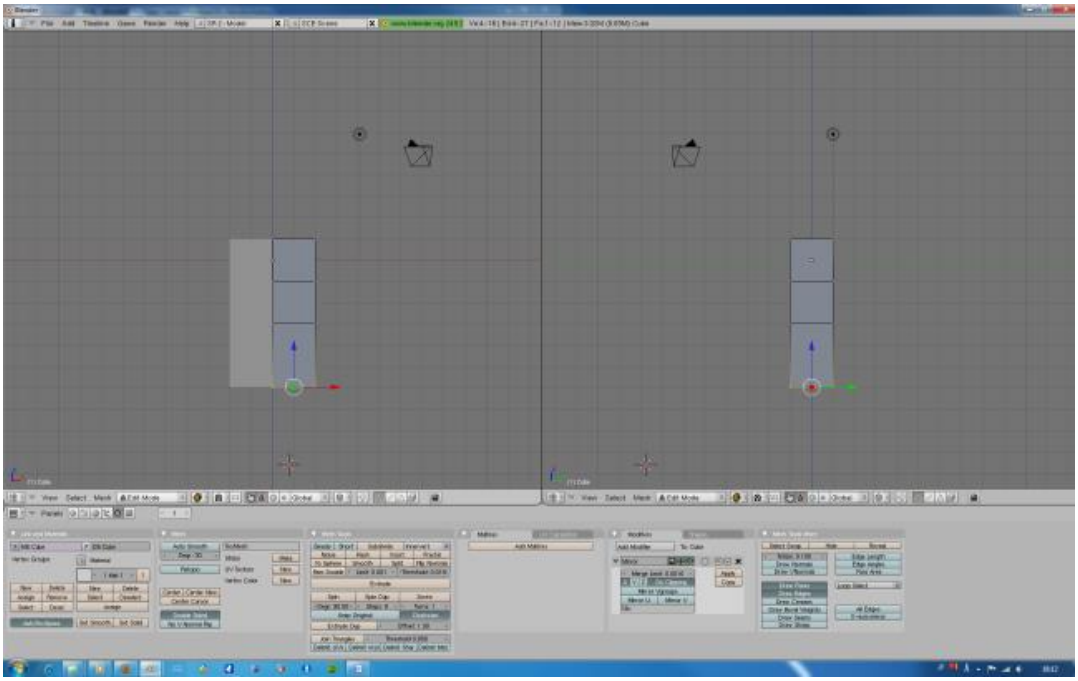

Imagen 13

Desmarcar el botón “Do Clipping” en la pestaña “modifiers” para que la cara del modelo que está situada sobre el eje Y se separe de ella. Ahora, sin dejar de seleccionar la cara inferior, pulsamos la tecla “G” y desplazamos hacia la derecha para abrir un poco la pierna (Imagen 14).

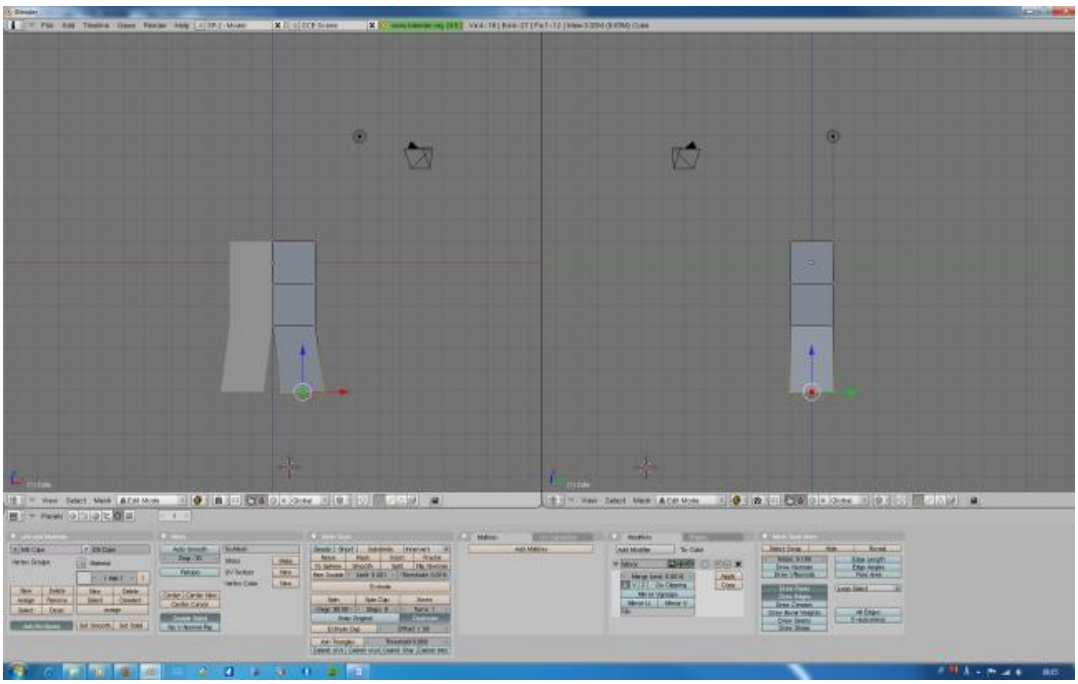

Imagen 14

Volvemos a extrudir para completar la pierna. En la vista lateral, seleccionamos los vértices de la rodilla con la tecla “B”. A continuación, en la vista lateral, pulsamos la tecla “G” y movemos el ratón hacia la izquierda para flexionar un poco las rodillas (Imagen 15).

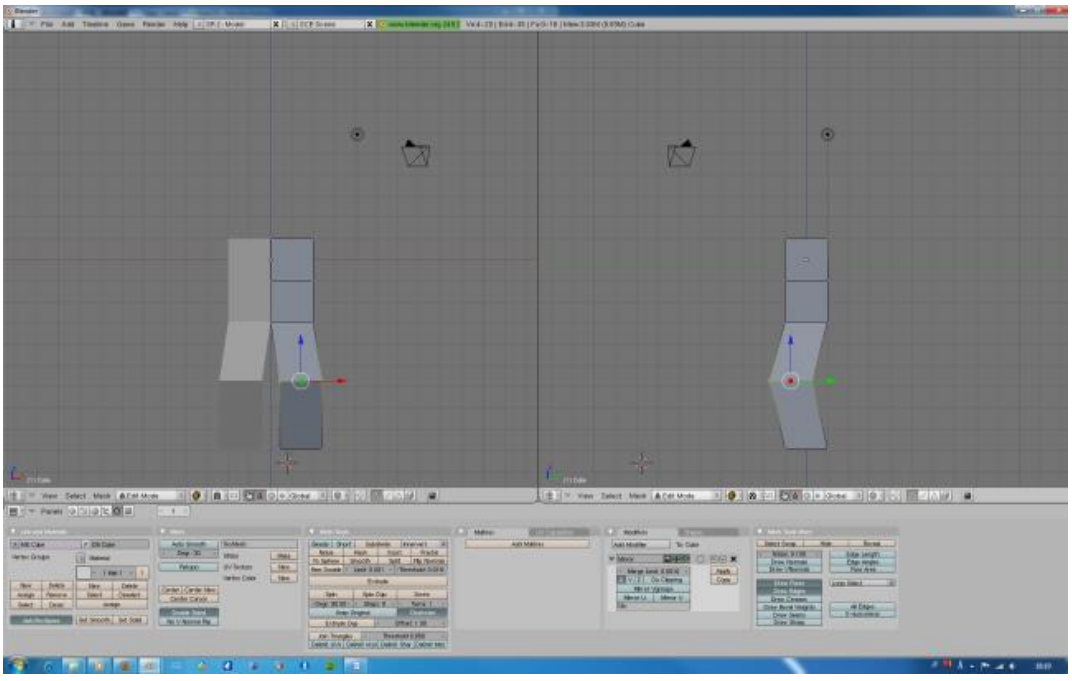

Imagen 15

Volver a extrudir para crear el pie (Imagen 16).

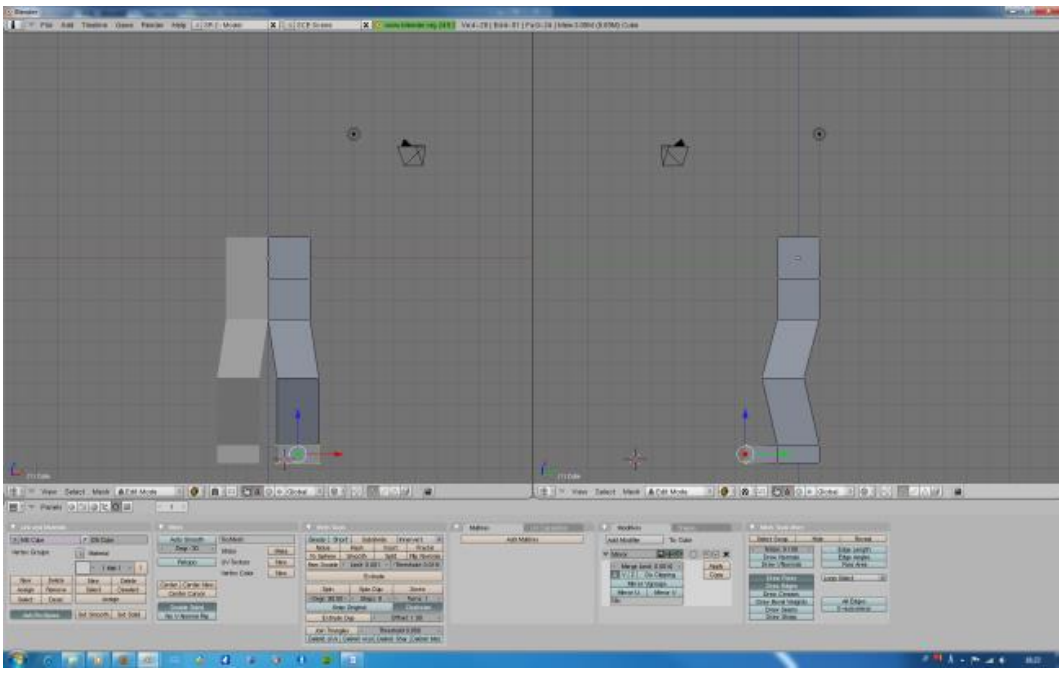

Imagen 16

Habilitar 'Do Clipping' y extrudir el tronco para crear los hombros (Imagen 17).

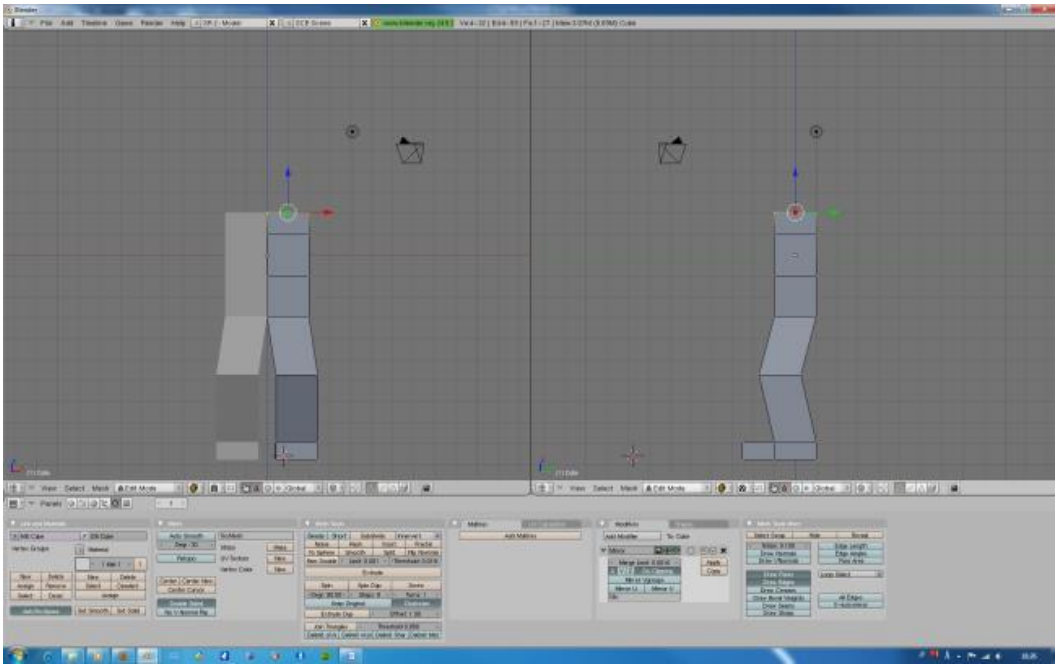

Imagen 17

Seleccionar la cara derecha de la región extruida. Extrudir dos veces esta cara para crear el brazo (Imagen 18).

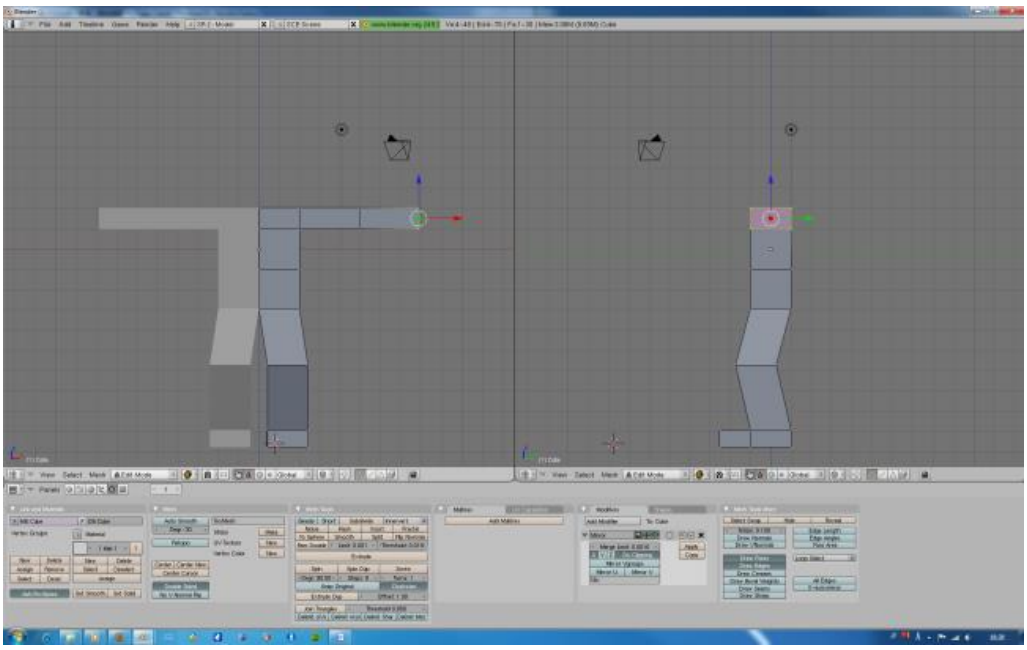

Imagen 18

Seleccionar los vértices del codo y trasladar sobre el eje Y hacia atrás para doblar un poco el brazo. Para ello pulsar 'G' + 'Y' en la vista lateral y desplazar hacia la derecha (Imagen 19).

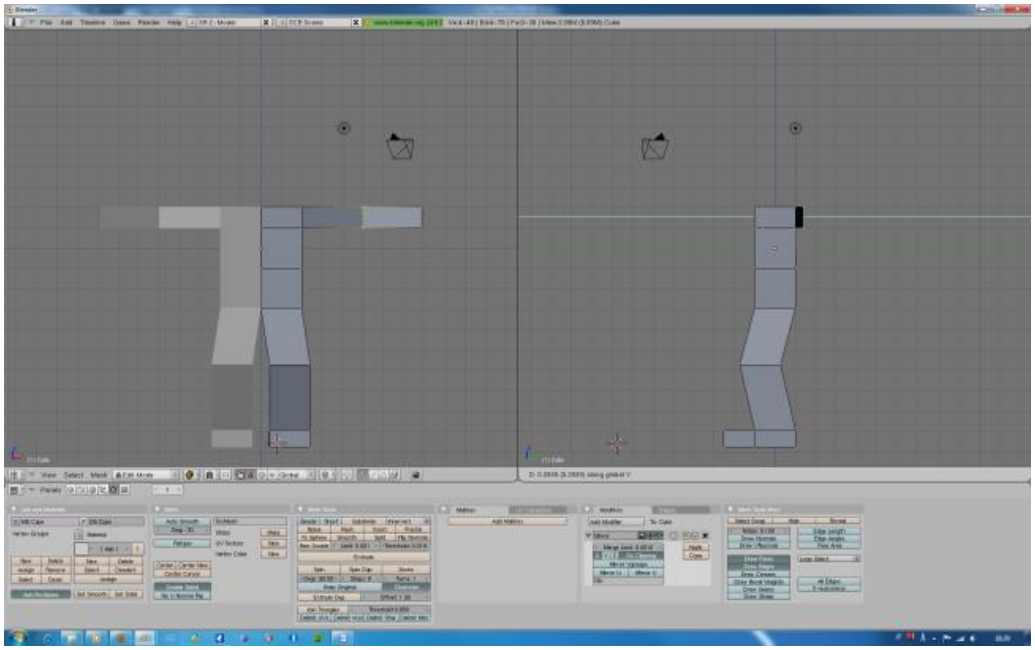

Imagen 19

Seleccionar 'View' -> 'Top' para ver el cuerpo desde arriba (Imagen 20).

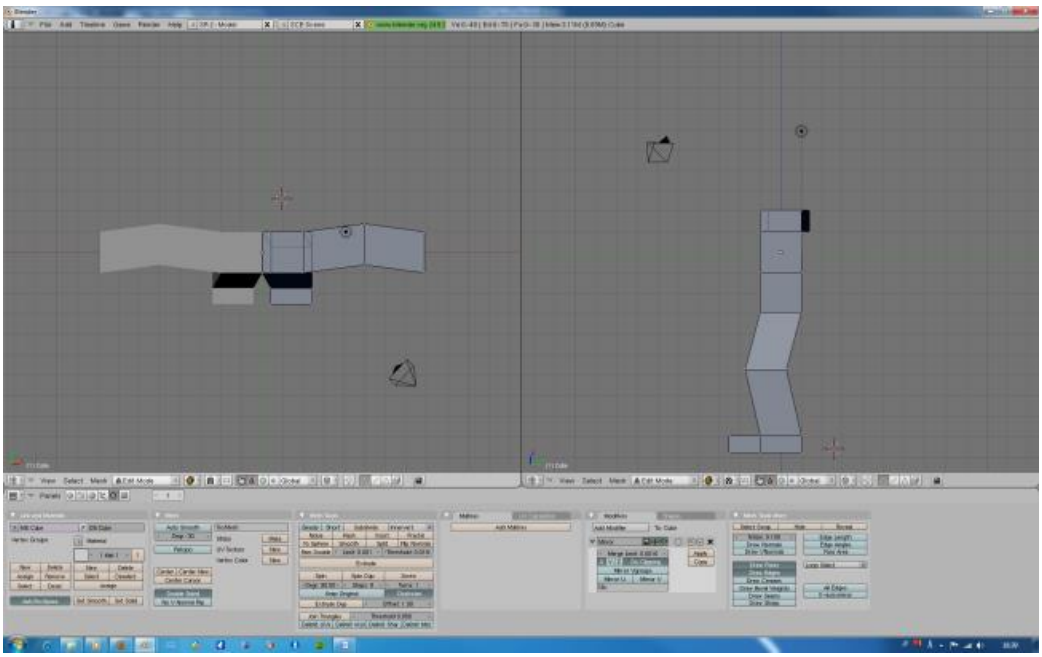

Imagen 20

Seleccionar 'Face select mode' (Imagen 21).

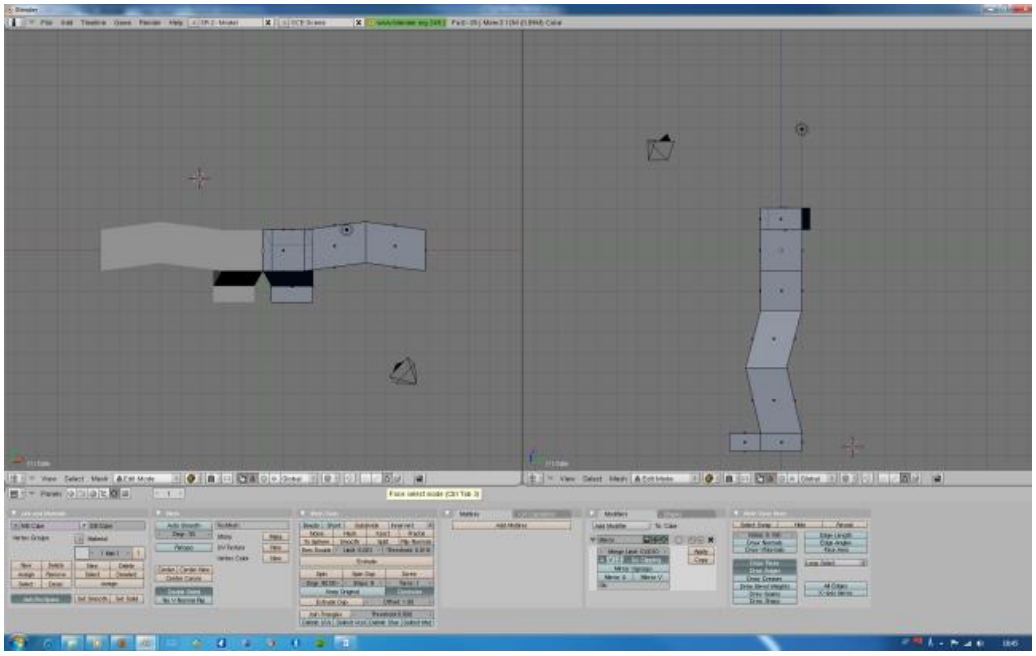

Imagen 21

Seleccionar la cara del hombro (Imagen 22).

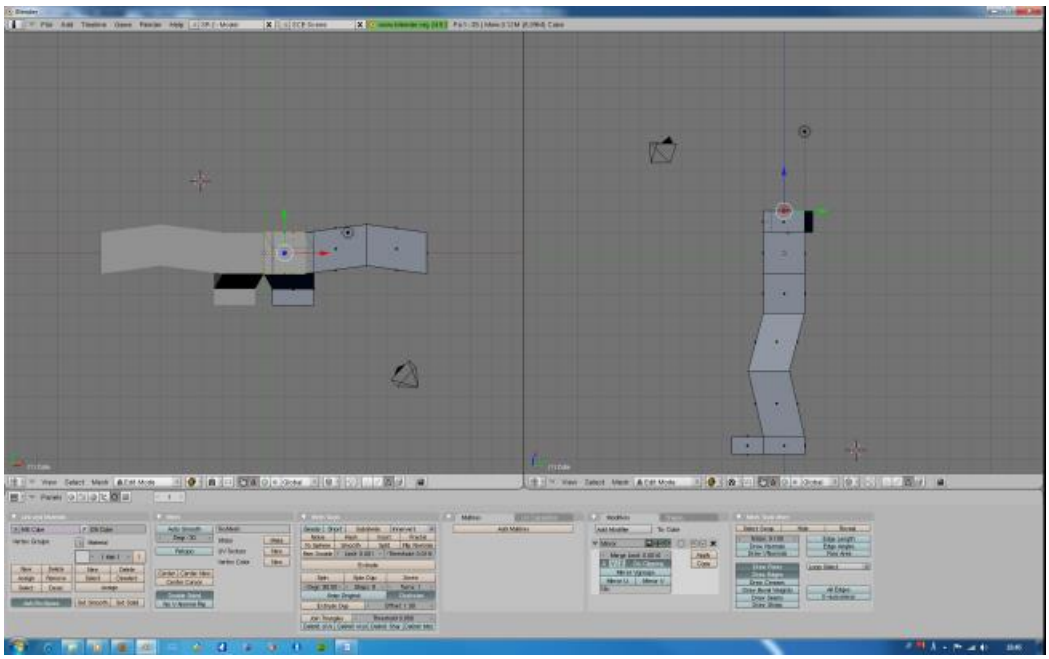

Imagen 22

Pulsar espacio para sacar el menú de acciones. Seleccionar 'Edit' -> 'Edges' -> 'Subdivide' (Imagen 23).

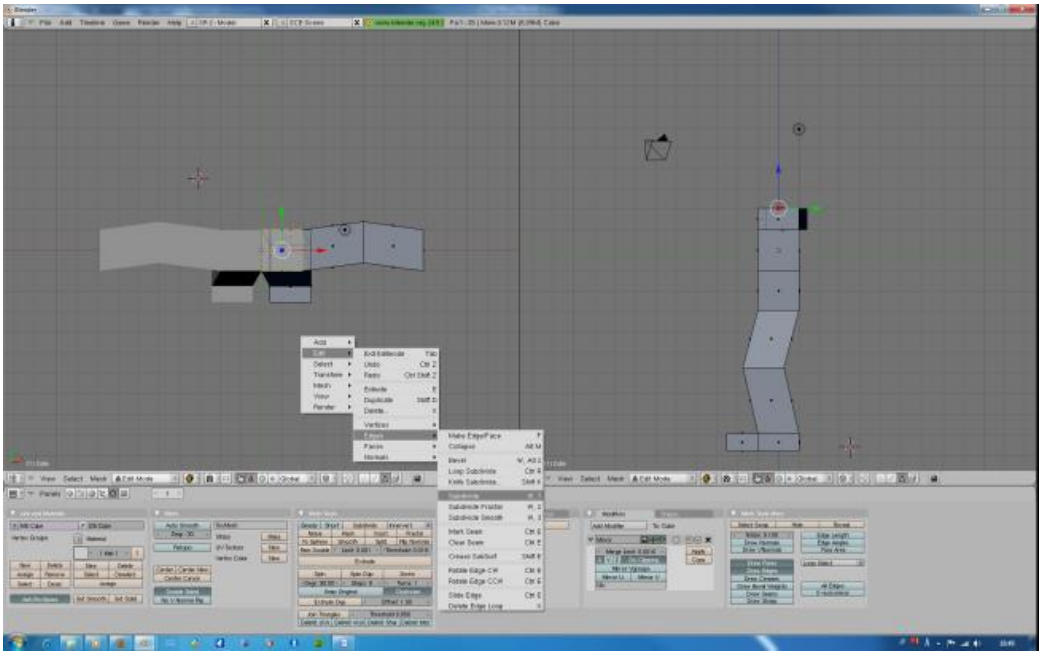

Imagen 23

Seleccionar los dos cuadrados de la izquierda que se han generado, y volver a dividir (Imagen 24).

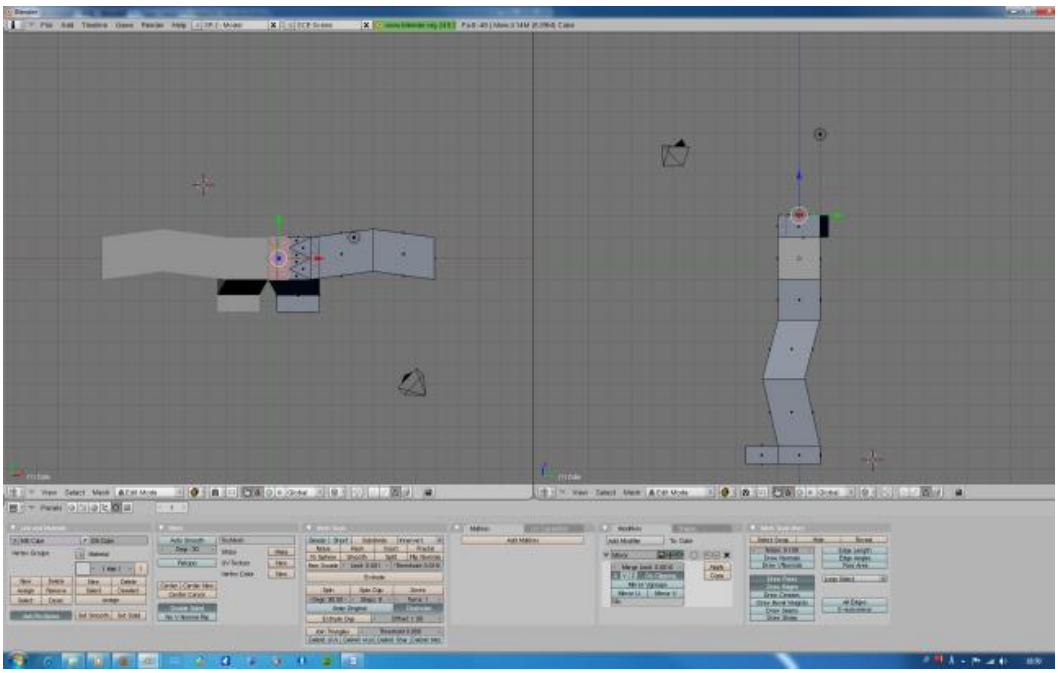

Imagen 24

Seleccionar los dos cuadrados centrales para extrudir y crear el cuello (Imagen 25).

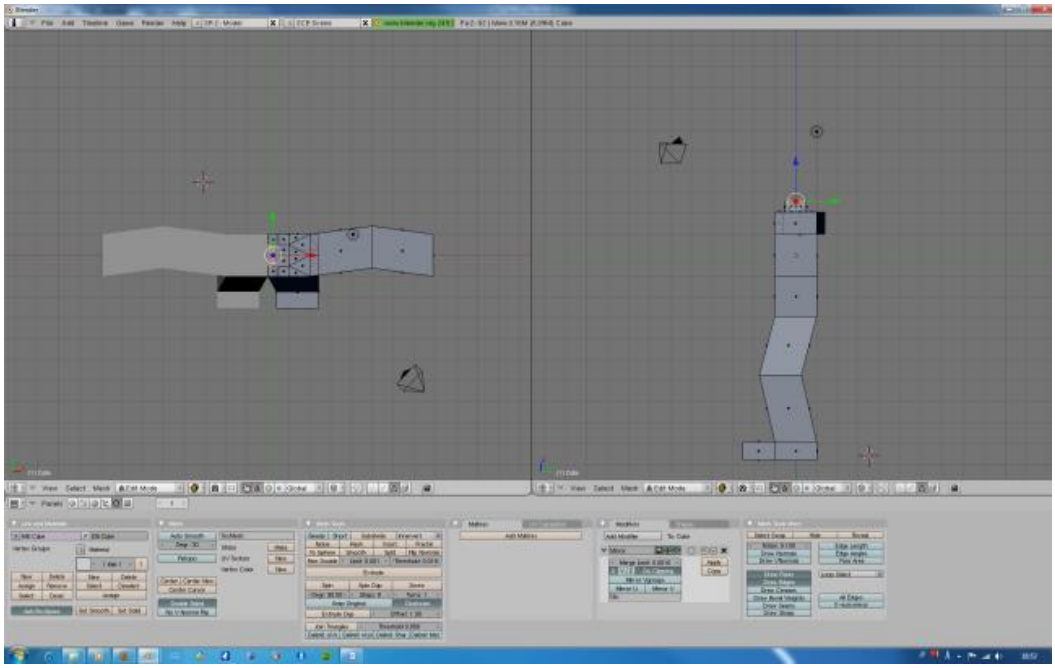

Imagen 25

Volver a extrudir el cuello para crear la cabeza. Escalar la cara a la vez que se extrude y crear la cabeza (Imagen 26).

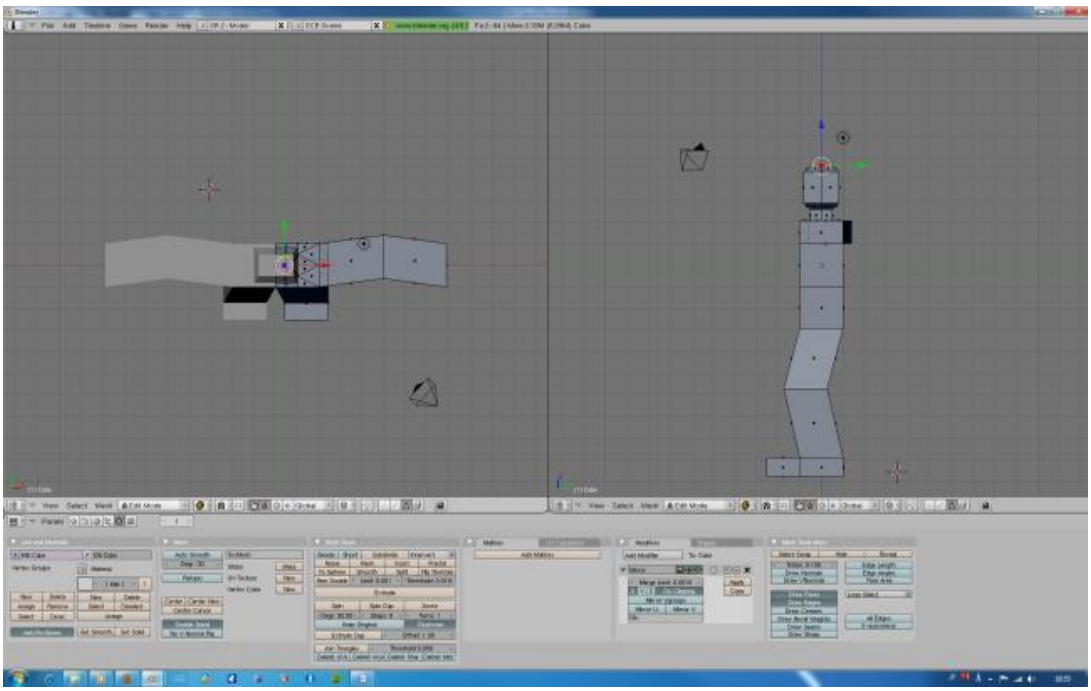

Imagen 26

Una vez creada la forma, se añaden los detalles y se corrigen los vértices para obtener el modelo deseado. Y después de algunas limpiezas obtenemos el siguiente resultado (Imagen 27):

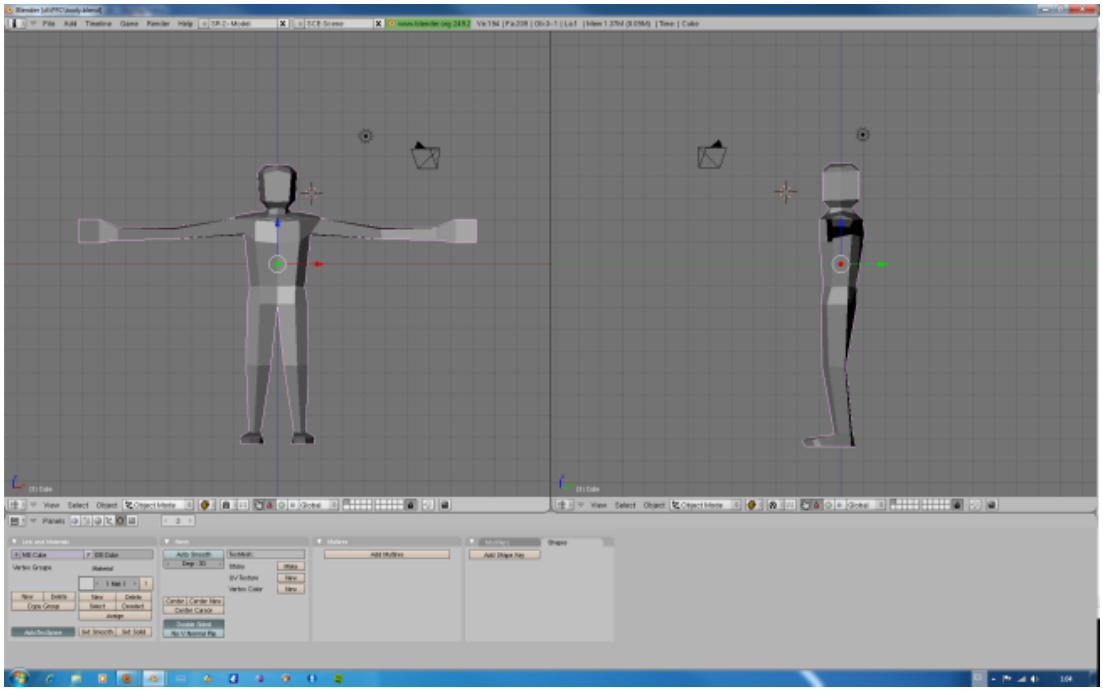

Imagen 27

Como nos ha quedado el brazo demasiado largo lo vamos a reducir. Seleccionamos los vértices del brazo (B+Rueda ratón) y los reducimos escalando sobre el eje X. El resultado es el siguiente (Imagen 28):

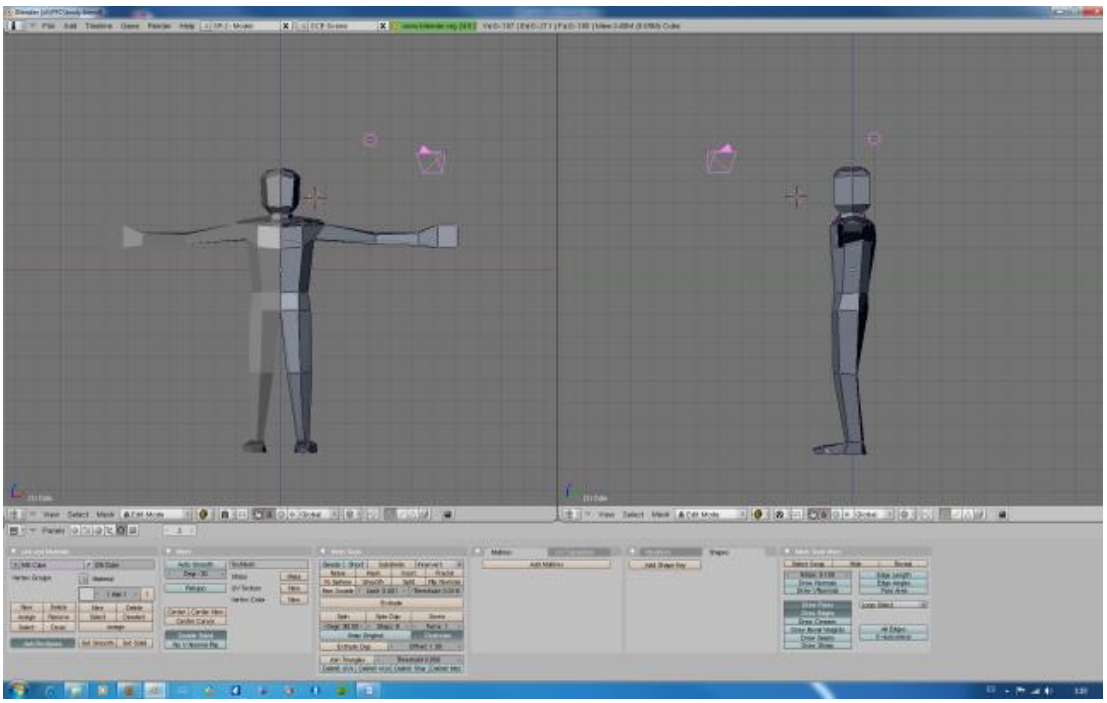

Imagen 28

Con esto tenemos la base del avatar, ahora nos queda dividir los polígonos hasta el nivel de detalle que creamos oportuno. En este caso, como lo vamos a utilizar en un simulador en "tiempo real", procuraremos que la malla no tenga más de 3000 polígonos (triángulos) para que se pueda renderizar sin retardos en los movimientos del avatar. Si lo modelamos con muy alto nivel de detalle como en las películas de animación en 3D, el renderizado de cada fotograma es mucho más lenta y no es viable para un sistema de renderizado en tiempo real.

Después de unos cuantos refinamientos obtenemos el siguiente aspecto (Imagen 29):

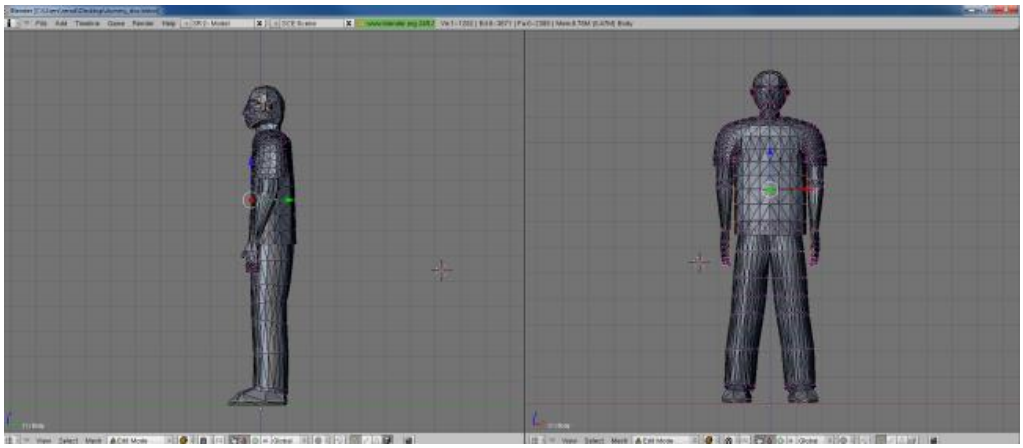

Imagen 29

Una vez terminado aplicamos el efecto espejo para materializar la otra mitad del cuerpo, pulsando el botón "apply" en la pestaña "Modifiers" (Imagen 30).

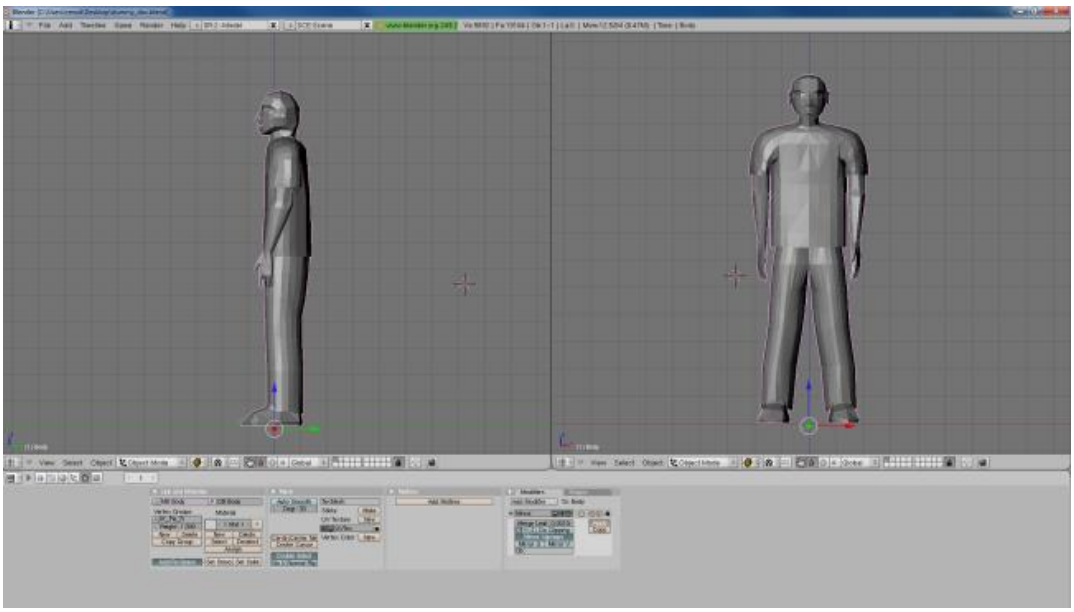

Imagen 30

A2.2 DEFINICIÓN DE LA TEXTURA

Creamos las costuras para que al desenvolver la textura quede bien extendida en un plano sin solapamientos.

Primero de todo, eliminamos una de las dos ventanas "3D View" para poder marcar mejor las costuras. Pulsamos el botón derecho del ratón en la franja de separación de los dos “3D Views” y seleccionamos “Join Areas” (Imagen 31):

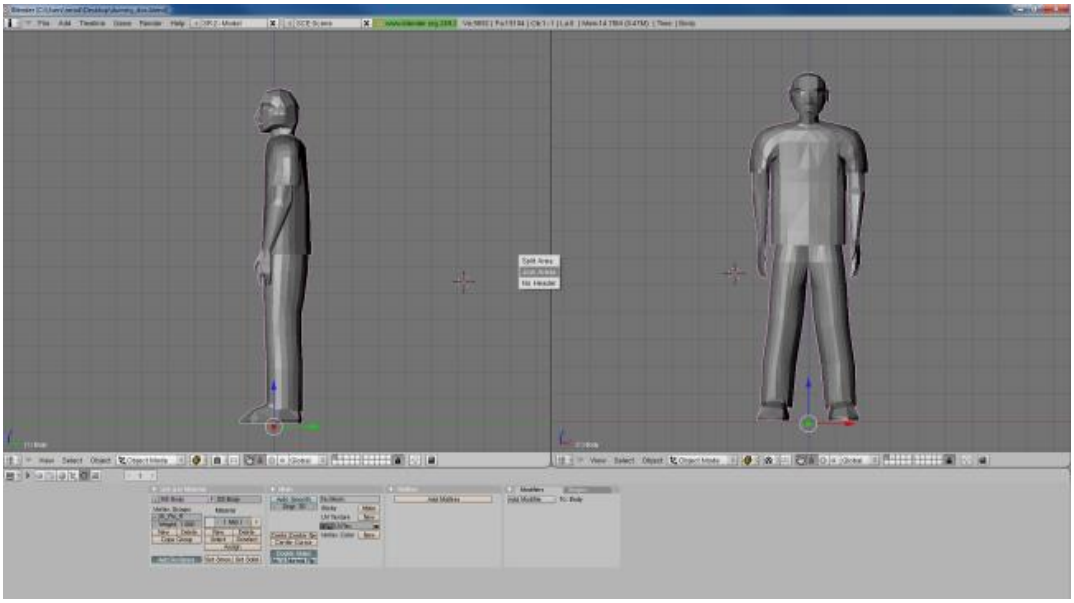

Imagen 31

Hacemos clic con el botón izquierdo del ratón sobre la ventana de la derecha (Imagen 32).

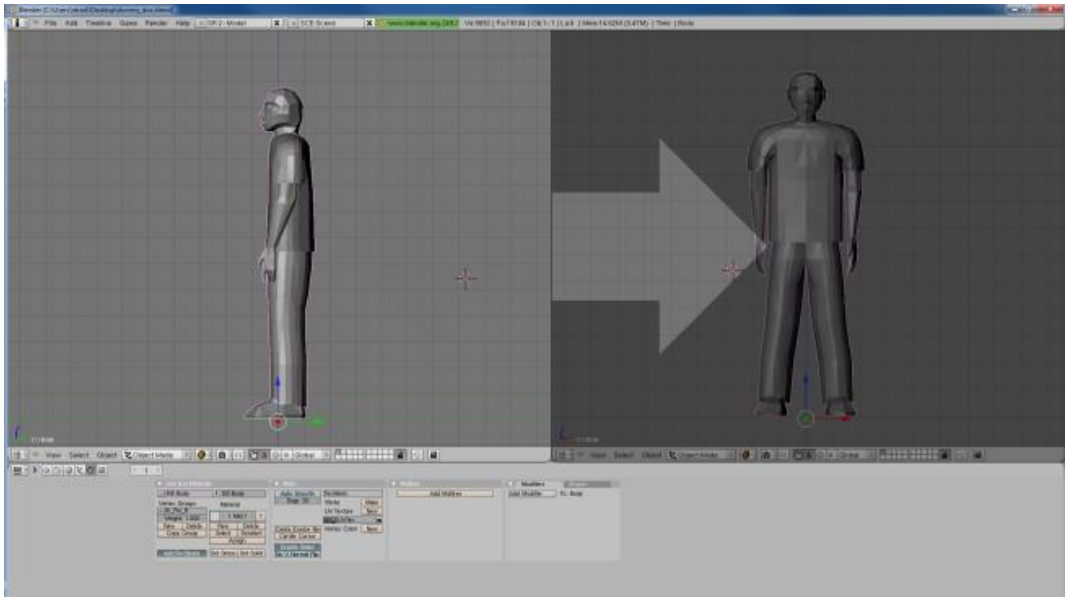

Imagen 32

Y ya tenemos un único “3D View” agrandado (Imagen 33).

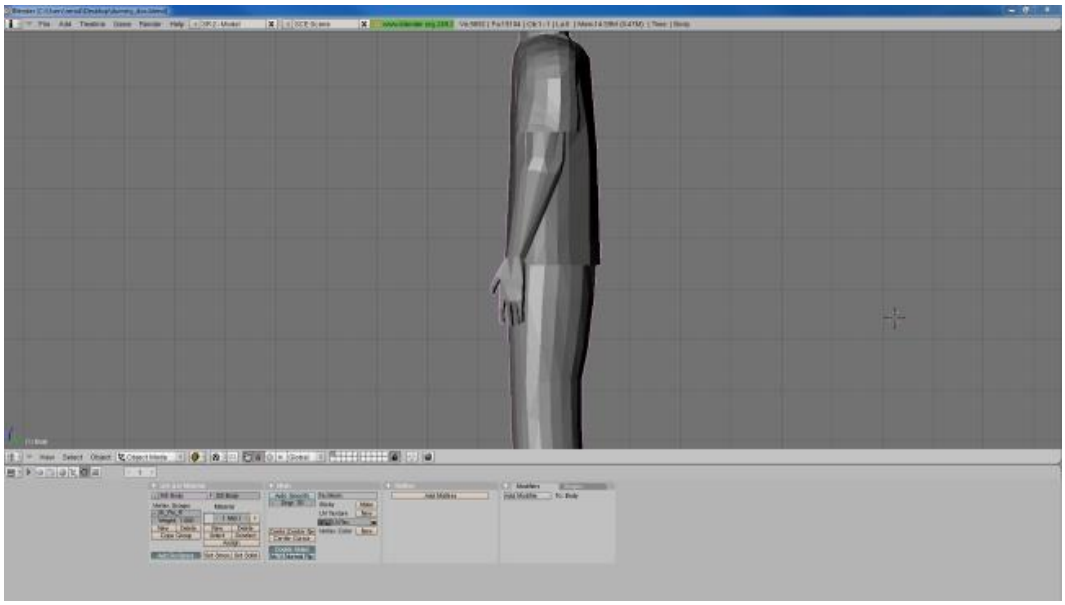

Imagen 33

Empezamos por los zapatos (Imagen 34):

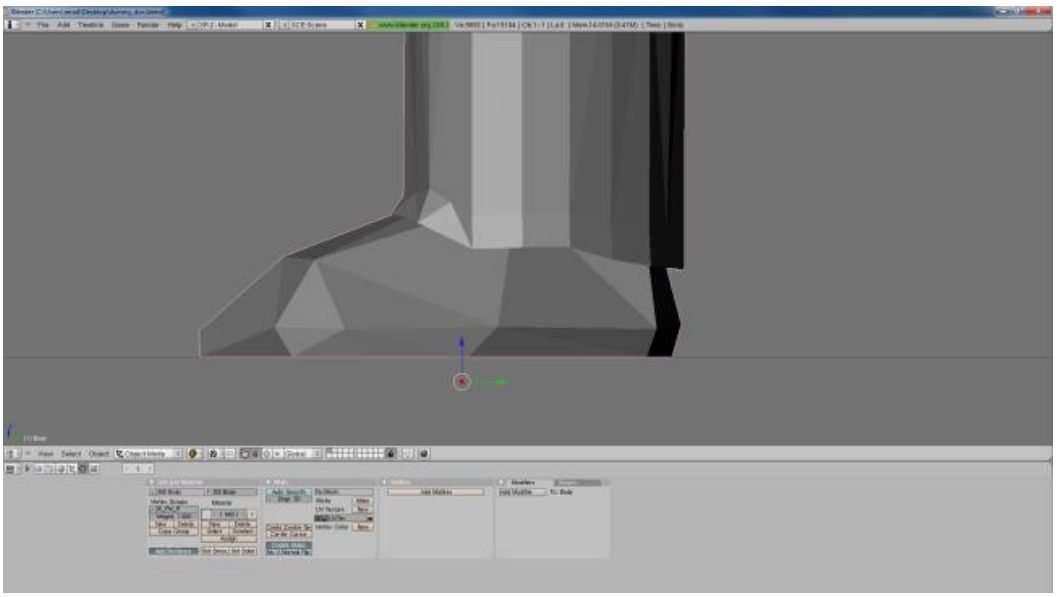

Imagen 34

Entramos en modo de edición (pulsamos Tabulador) y seleccionamos las aristas que deseamos que sean las costuras (Imagen 35):

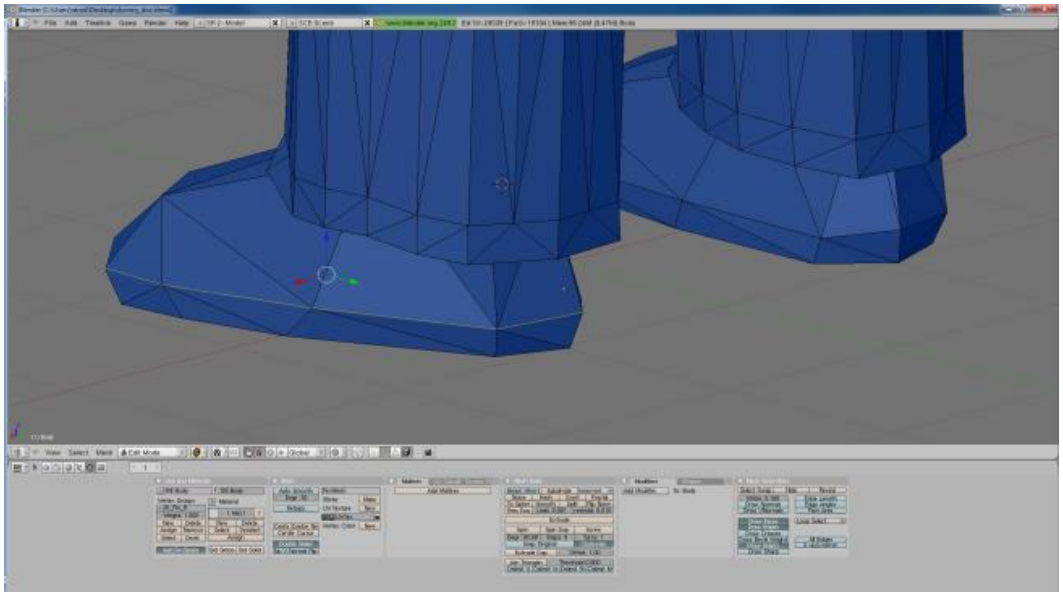

Imagen 35

Una vez seleccionado pulsamos Ctrl-E y elegimos la opción "Mark seam" (Imagen 36):

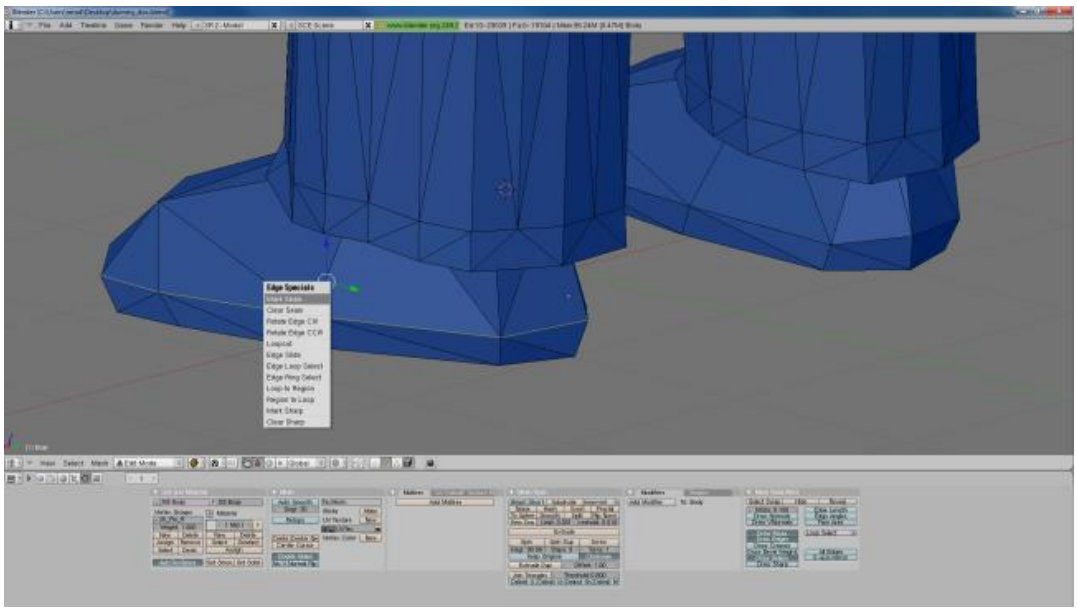

Imagen 36

Y se queda marcada en color naranja (Imagen 37):

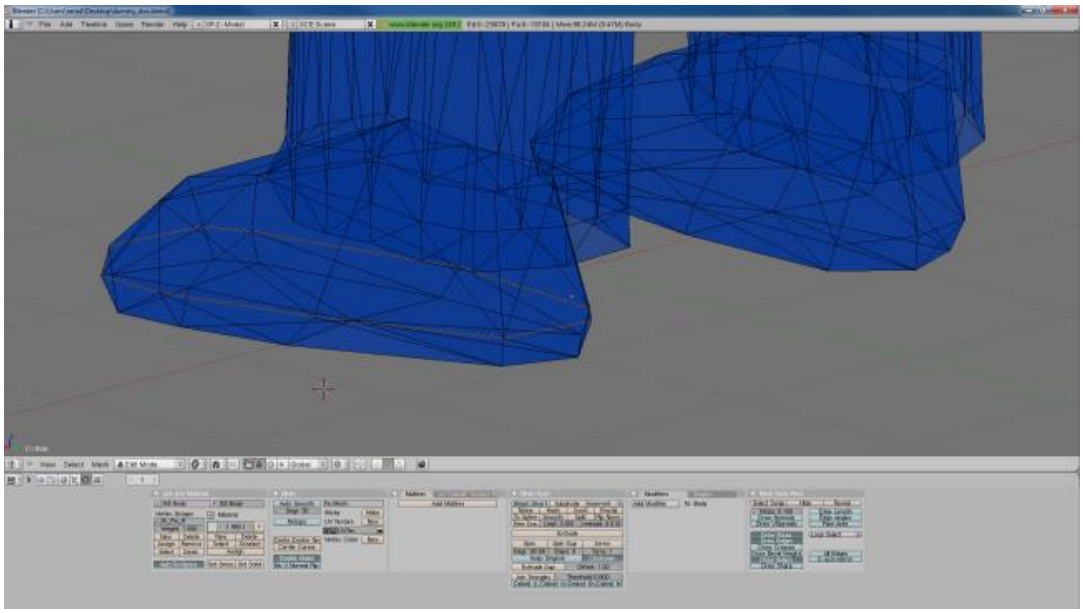

Imagen 37

Creamos las costuras en el otro zapato y en los pantalones también, y este es el resultado (Imagen 38):

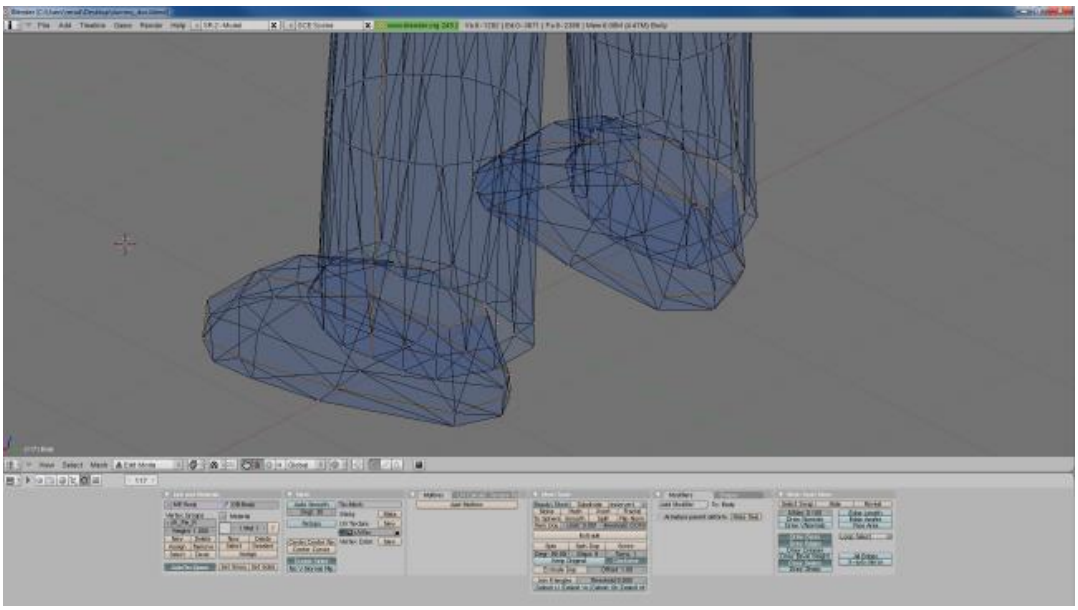

Imagen 38

También marcamos sobre la camiseta y en las manos (Imagen 39):

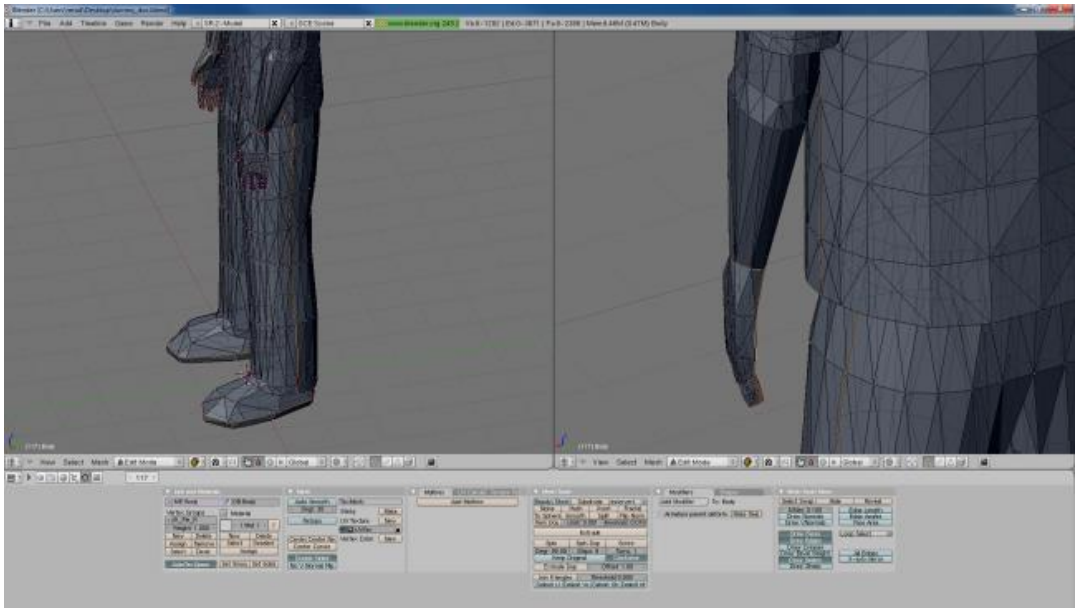

Imagen 39

Y también los brazos (Imagen 40):

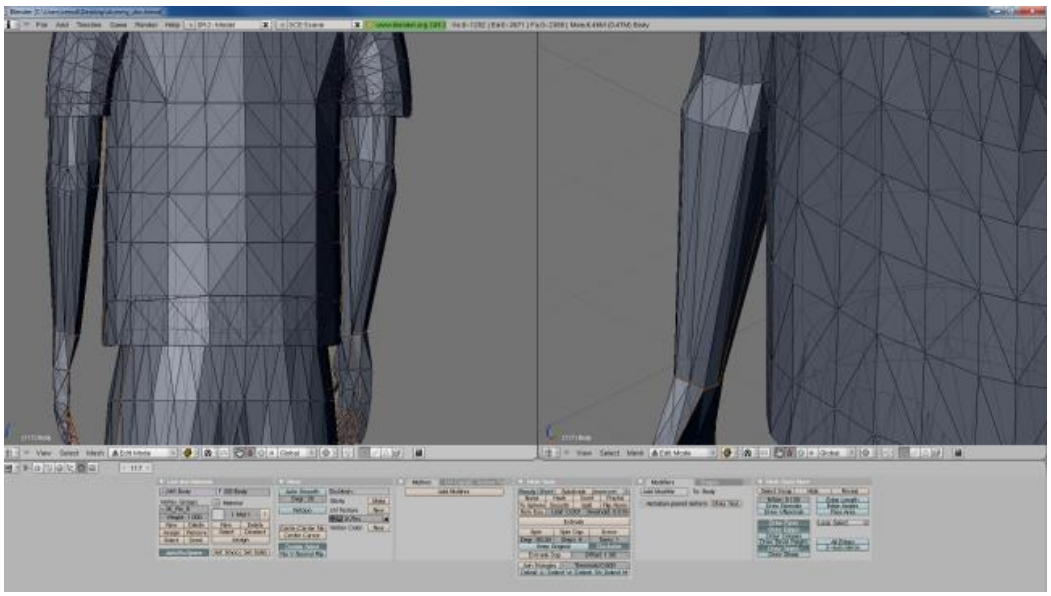

Imagen 40

Y para terminar en la cabeza (Imagen 41):

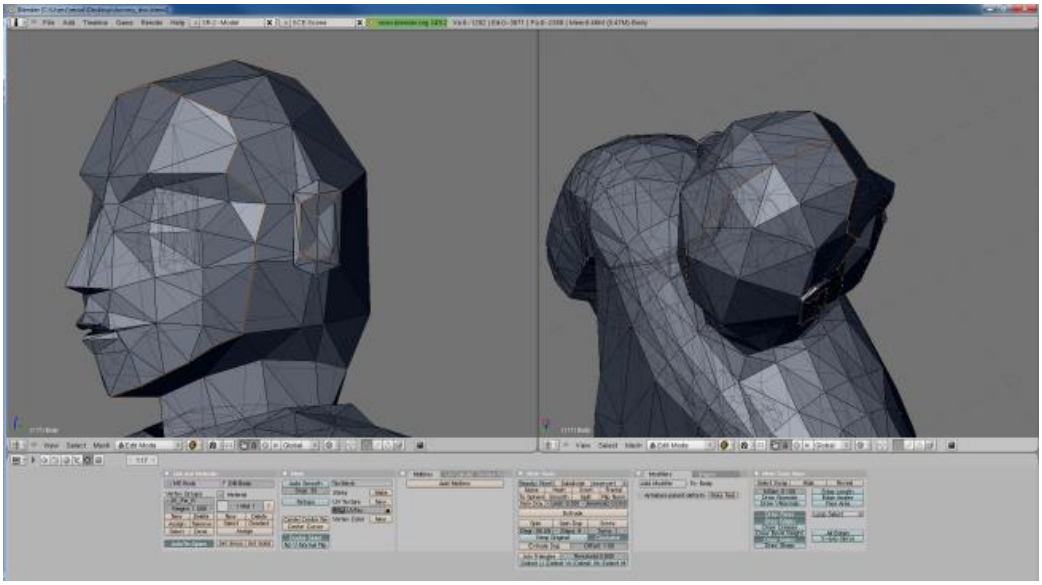

Imagen 41

Desenvolvemos las superficies, para ello cambiamos una de las ventanas a "UV/Image Editor" (Imagen 42):

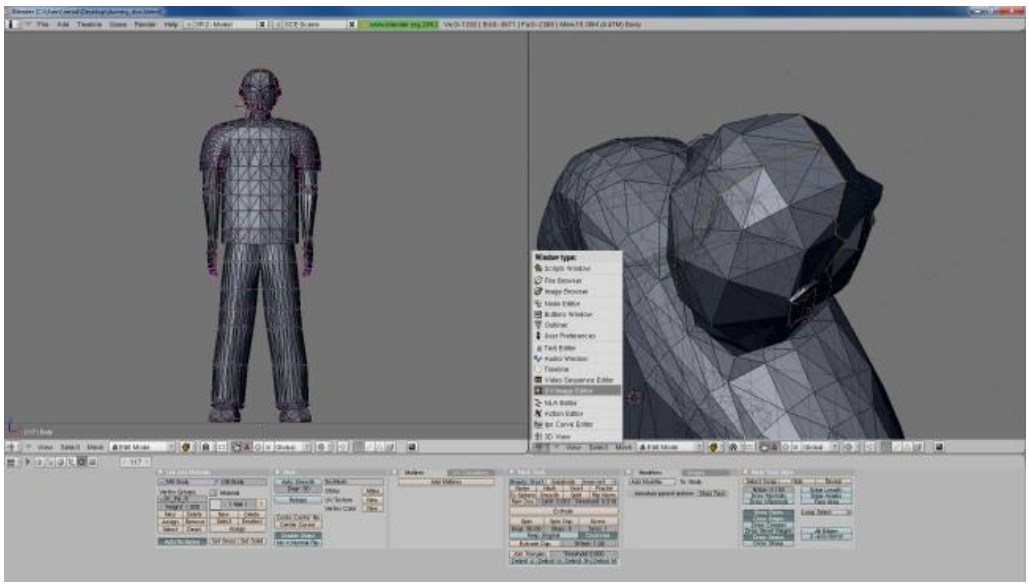

Imagen 42

Y nos aparece una rejilla en la ventana derecha (Imagen 43). En esta rejilla nos aparecerán posteriormente los vértices de la malla mapeados.

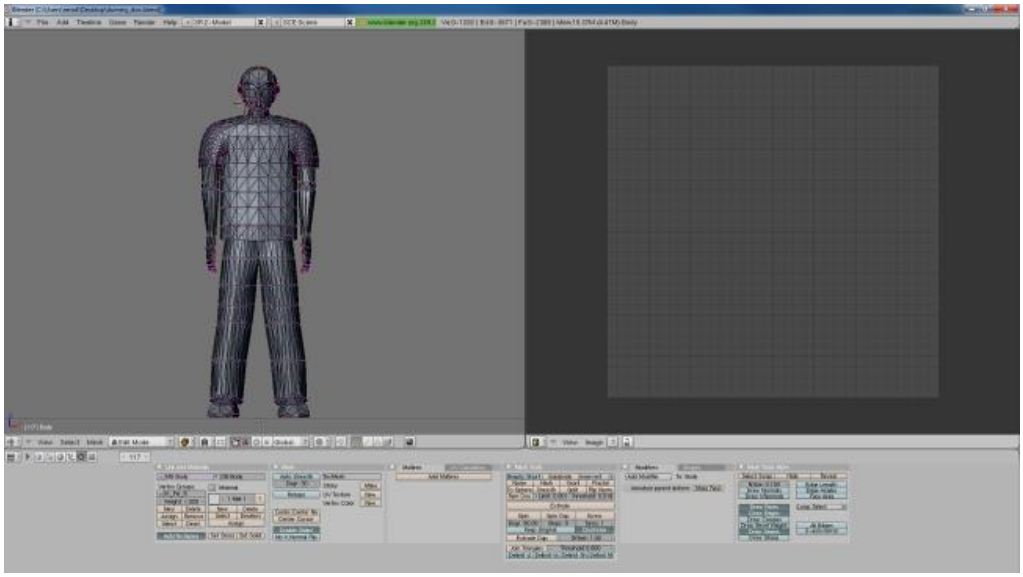

Imagen 43

Para desenvolver las superficies, primero pulsamos el botón "New" que está al lado de la etiqueta "UV Texture", en el área inferior dentro del cuadro "Mesh" (Imagen 44):

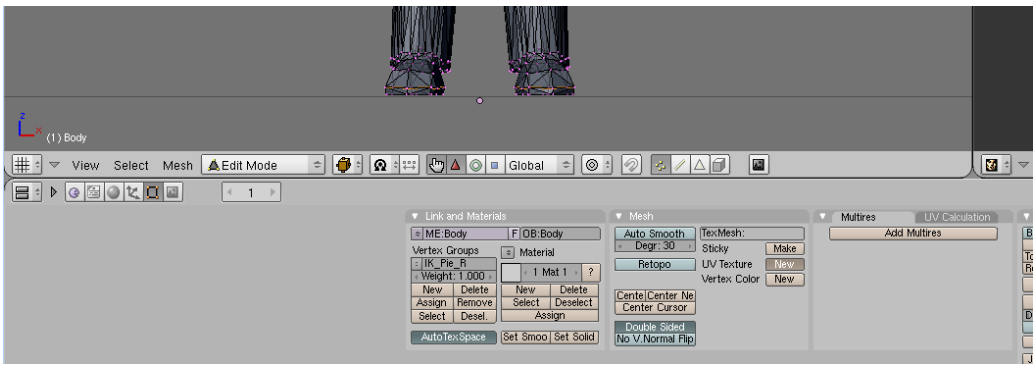

Imagen 44

Y nos aparece una nueva textura llamada UVText (Imagen 45):

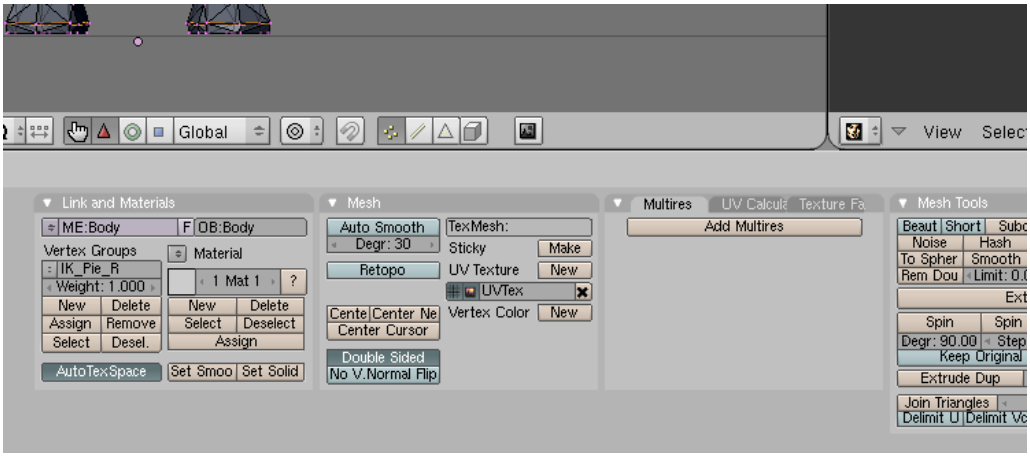

Imagen 45

Ahora seleccionamos todas las caras del avatar pulsando 'A' (Imagen 46):

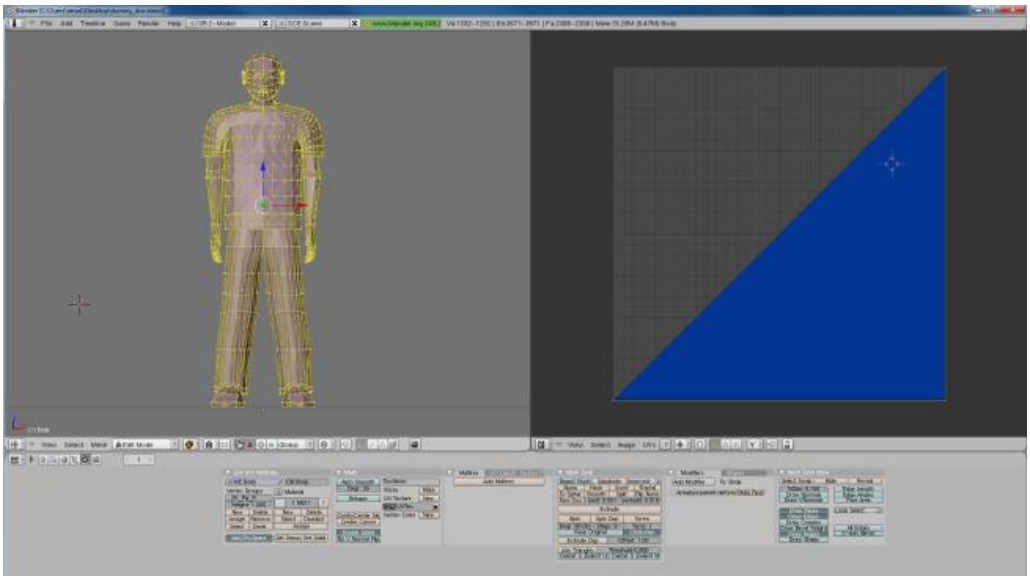

Imagen 46

Y en la misma ventana pulamos 'U' y seleccionamos "Unwrap" (Imagen 47):

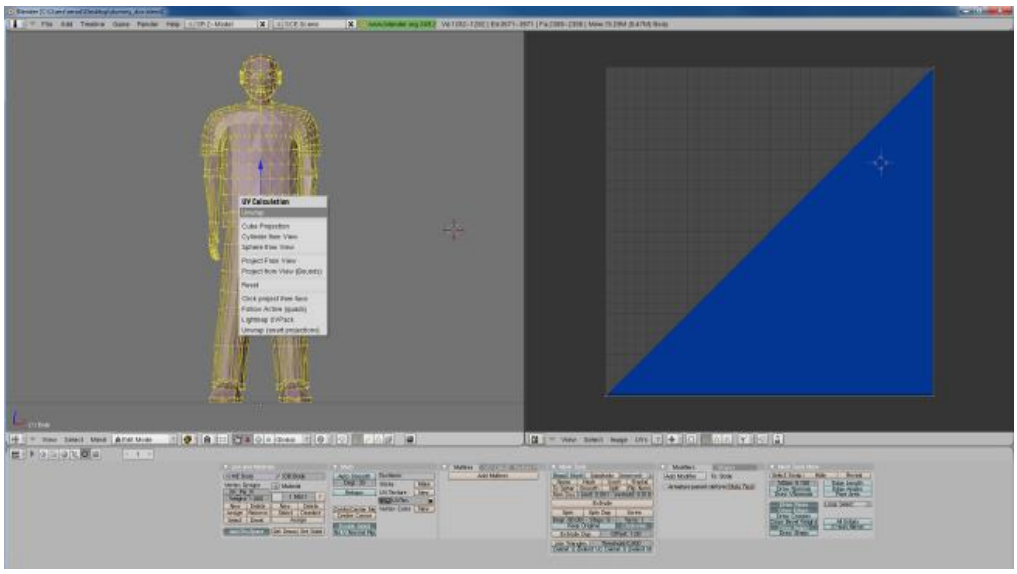

Imagen 47

Y nos aparecen todos los vértices de la malla mapeados sobre la rejilla (Imagen 48):

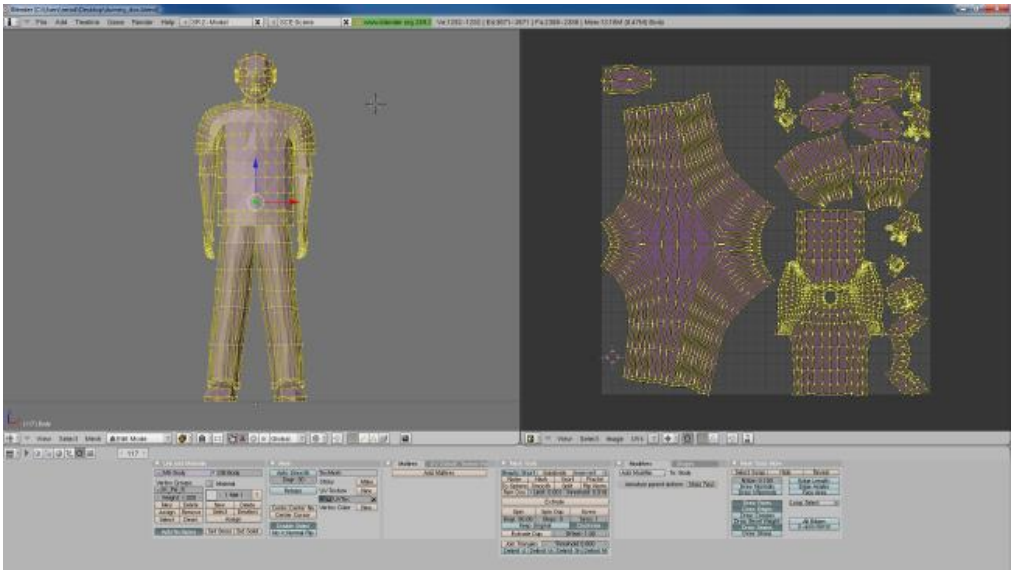

Imagen 48

Una vez desenvuelta, podemos manipular los vértices en el "UV Image Editor" de la misma forma que en "3D View": rotar, escalar, mover.

Vamos a organizar un poco las superficies para que sea más fácil identificar las partes del cuerpo a la hora de dibujar las texturas. Para ello es útil la combinación de teclas Ctrl+L que permite seleccionas un grupo de vértices conexos (tenemos que asegurarnos de que tenemos desactivado el botón en el "UV/Image editor", ver Imagen 49):

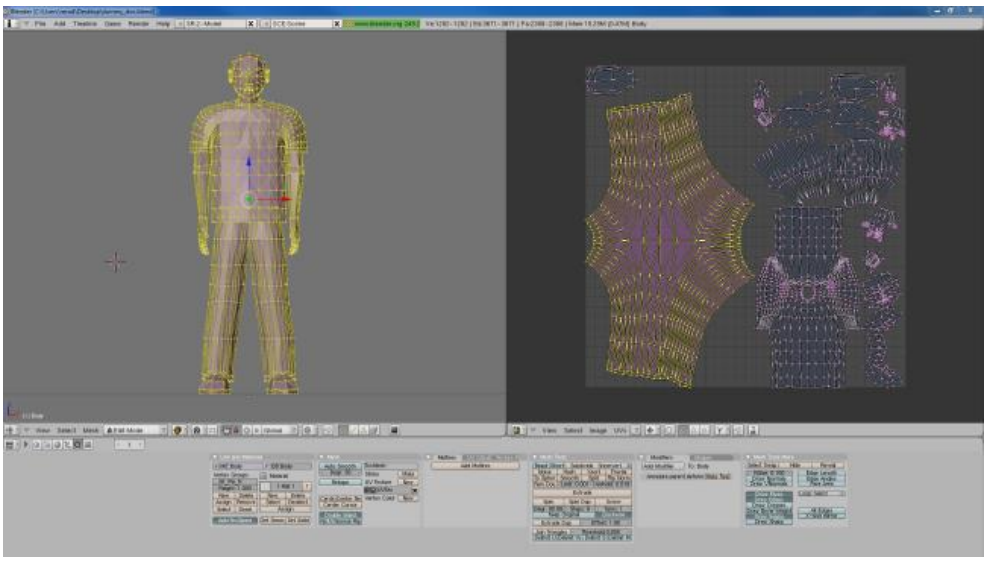

Imagen 49

Reorganizamos escalando, moviendo y rotando las superficies, y una vez hecho esto queda de la siguiente manera (Imagen 50):

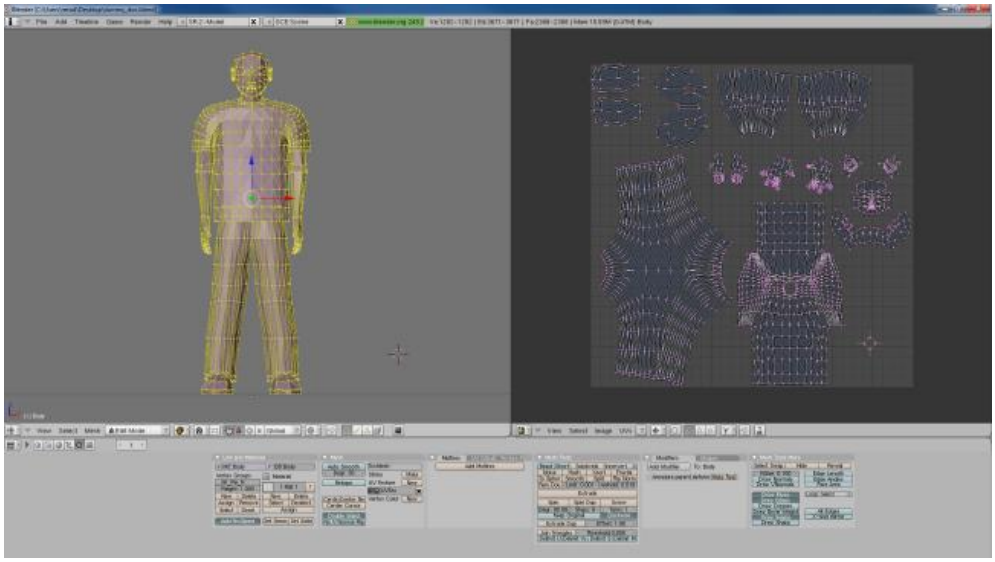

Imagen 50

Exportamos el layout a una imagen para poder pintar nuestra textura. Para ello vamos a "UVs" -> "Scripts" -> "Save UV Face Layout..." (Imagen 51):

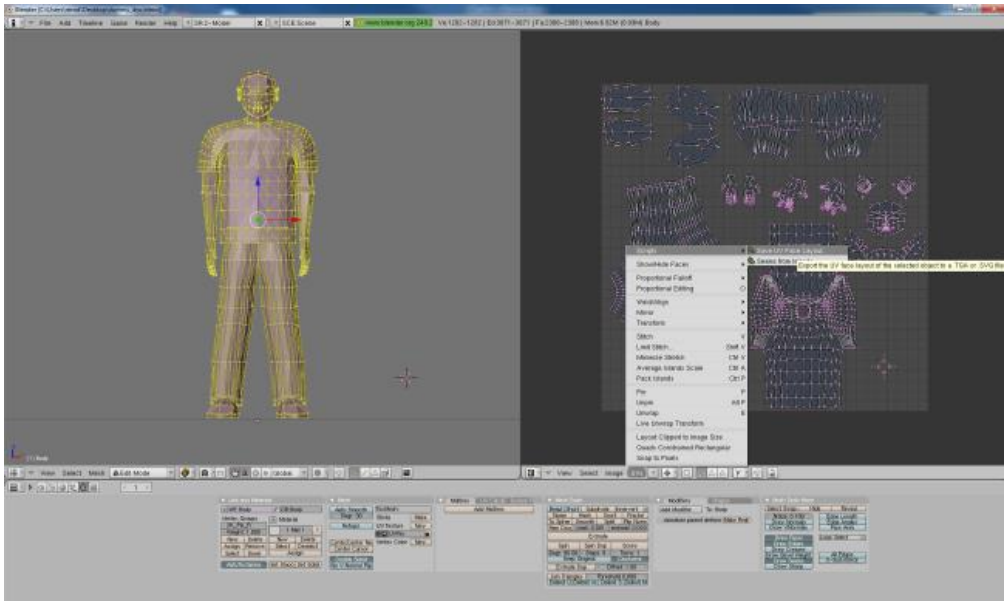

Imagen 51

Seleccionamos el tamaño de la textura, que en este caso es de 1024 pixeles, y también elegimos el grosor de los alambres que componen las superficies, en este caso "Wire:1" y pulsamos "OK" (Imagen 52):

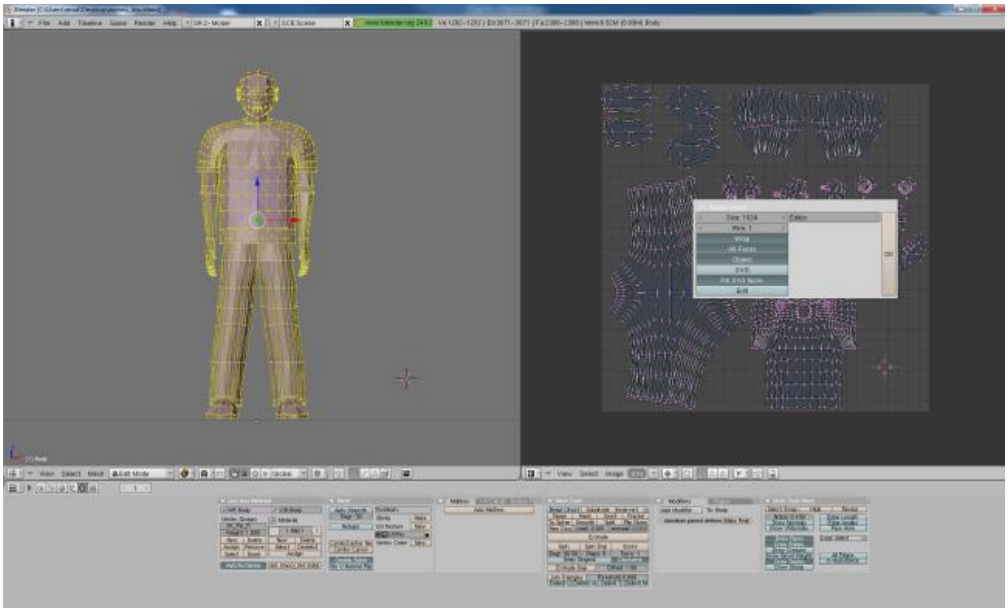

Imagen 52

Elegimos en el área inferior el directorio y el nombre del fichero y pulsamos "Save UV(tga)" (Imagen 53):

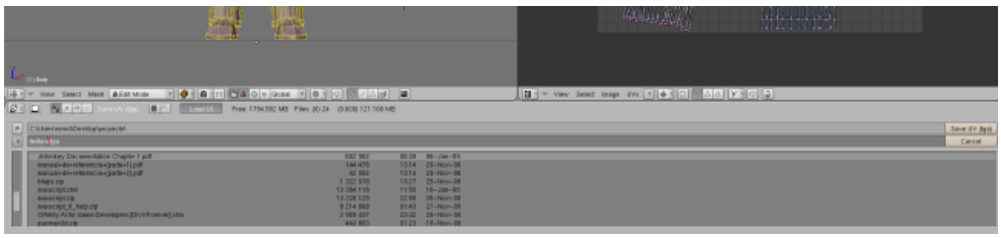

Imagen 53

A2.3 EDICIÓN DE LA TEXTURA CON GIMP

Abrimos el fichero exportado con GIMP (Imagen 54):

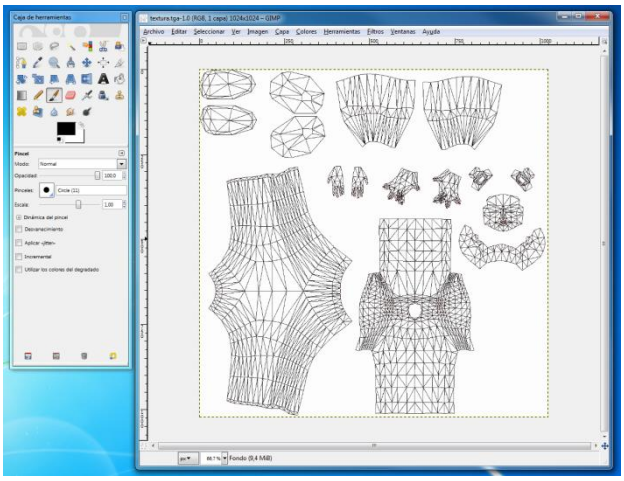

Imagen 54

Para empezar a dibujar creamos una capa transparente por encima de la imagen que hemos abierto y dibujamos en ella. De tal forma que la imagen exportada nos sirva de guía. Para ello, pulsamos "Ctrl+L" para que nos aparezca la ventana de capas (Imagen 55):

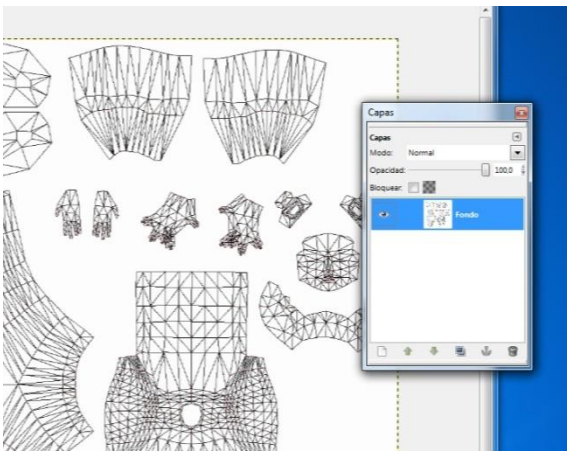

Imagen 55

A continuación, creamos la capa transparente pulsando sobre , situado en la esquina inferior izquierda, en la ventana de capas. Nos aparecerá la siguiente ventana (Imagen 56):

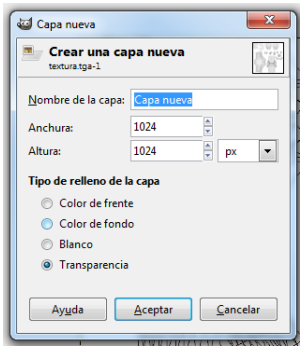

Imagen 56

Ponemos 1024 pixeles de anchura, 1024 pixeles de altura. Seleccionamos "Transparencia" como tipo de relleno de la capa y pulsamos "Aceptar".

Se nos añadirá la nueva capa en la ventana de capas (Imagen 57):

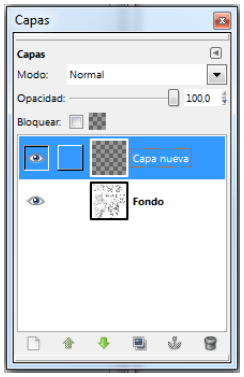

Imagen 57

Coloreamos las superficies guiándonos en la imagen de la capa inferior (Imagen 58):

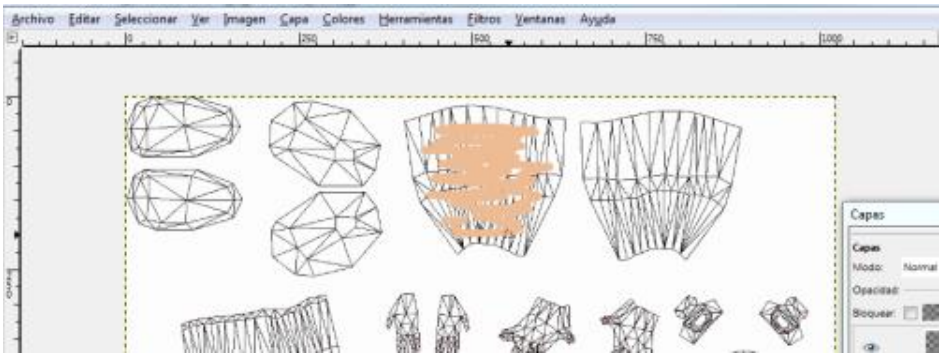

Imagen 58

Tras colorear todo nos queda la capa de la siguiente forma (Imagen 59):

Imagen 59

Y con respecto a la capa inferior queda de la siguiente forma (Imagen 60):

Imagen 60

Vamos a añadir algunos detalles como texturas y sombras. Podemos dibujar las sombras directamente en la textura para que el simulador no tenga que calcularlas en tiempo real. Aunque la sombra dibujada no es tan real como la calculada, se puede conseguir un aspecto visual más que aceptable, consiguiendo a su vez, reducir el tiempo de cálculo en el simulador.

La sombra la podemos generar en Blender mediante *texture baking*, que detallaremos los pasos a continuación.

Antes de generar las sombras redondeamos un poco la superficie del avatar aplicando un efecto llamado "Subsurf".

Primero, pulsamos sobre el botón "Add modifier", y a continuación seleccionamos la opción "Subsurf" (Imagen 61):

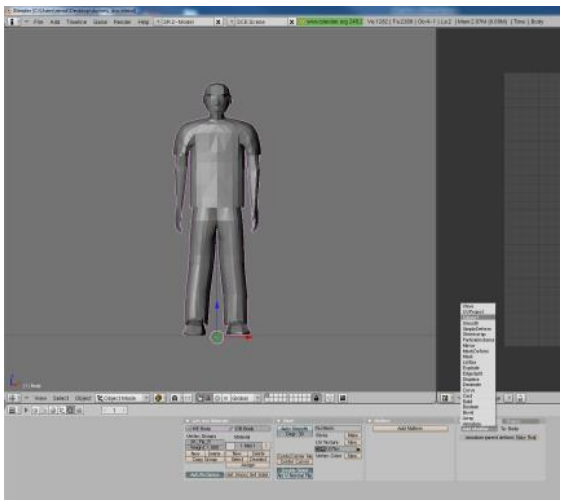

Imagen 61

Aumentamos la propiedad "Levels" hasta obtener el suavizado deseado, en este caso "Levels: 3" (Imagen 62):

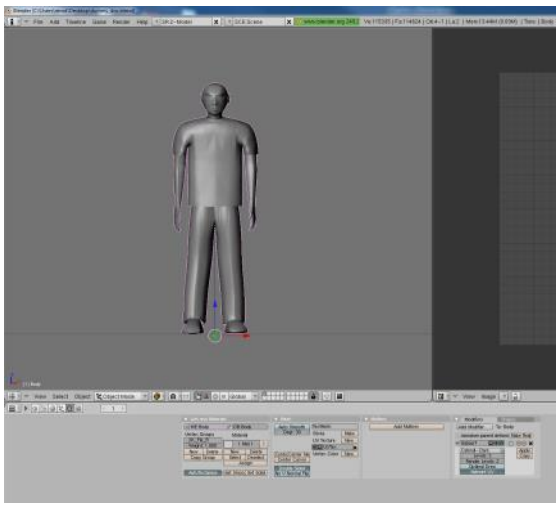

Imagen 62

Se puede observar que la superficie de la malla está más redondeada que antes.

Ahora aplicamos la técnica texture baking. Para ello, seleccionamos la malla y entramos en modo edición pulsando "Tabulador", y seleccionamos todas las caras pulsando "A" (Imagen 63):

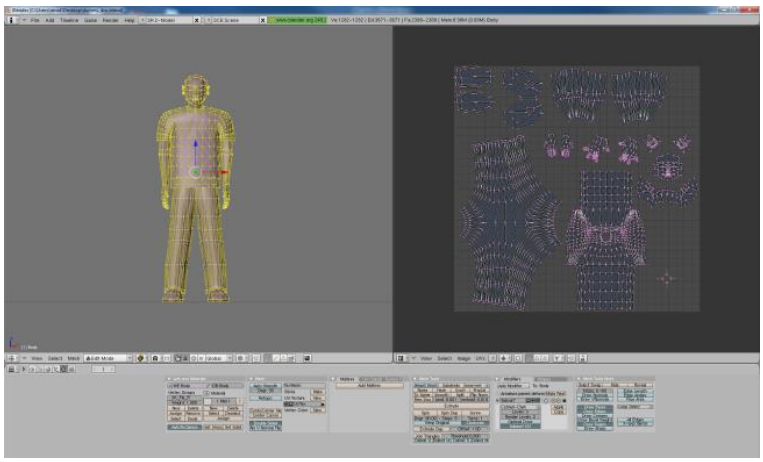

Imagen 63

En la ventana "UV/Image editor" abrimos el menú "Image" y seleccionamos la opción "New" para crear una nueva imagen de textura (Imagen 64):

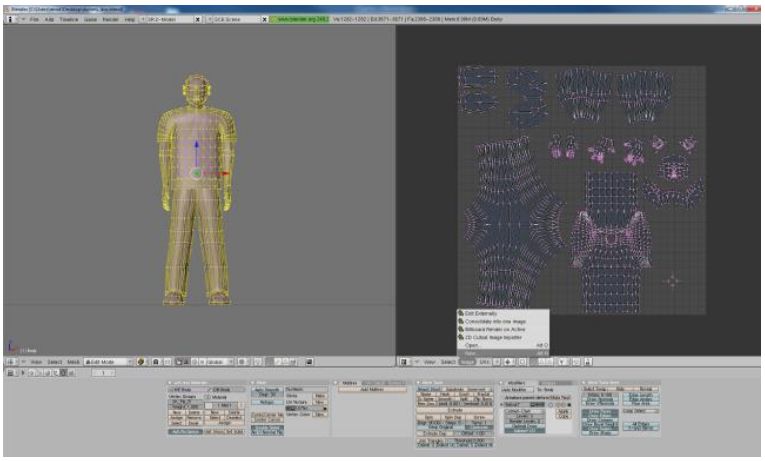

Imagen 64

Elegimos la anchura 1024 pixeles, altura 1024 pixeles, y escogemos el color de fondo y pulsamos "OK". En este caso el color de fondo escogido es el blanco (Imagen 65):

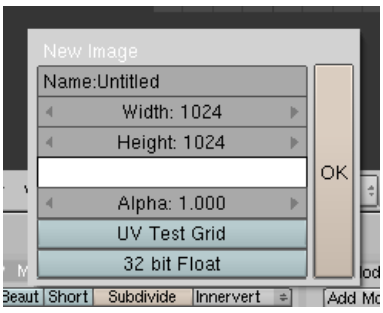

Imagen 65

Y nos dibujar la siguiente imagen (Imagen 66):

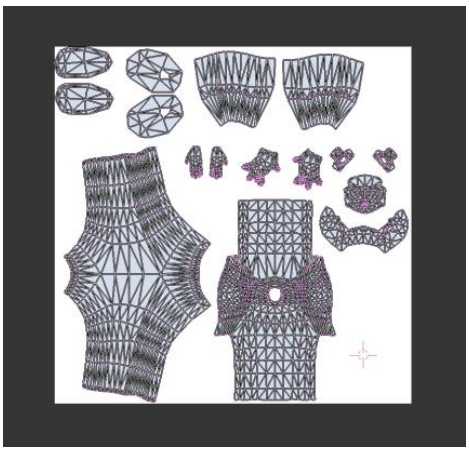

Imagen 66

Una vez hecho esto, vamos a calcular y pintar las sombras encima de la imagen que acabamos de crear. Pulsamos "F10" para ir a la opción "Scene" del área inferior y en la pestaña "Bake" seleccionamos "Ambient Occlusion" (Imagen 67):

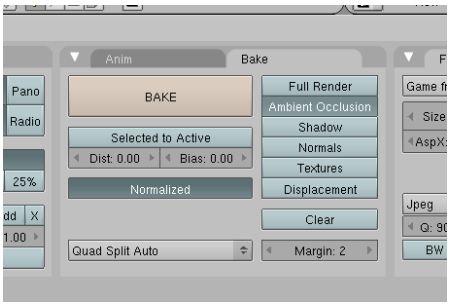

Imagen 67

Pulsamos sobre el botón "BAKE", y Blender empezará a calcular las sombras y las irá pintando sobre la imagen blanca que hemos creado anteriormente (Imagen 68):

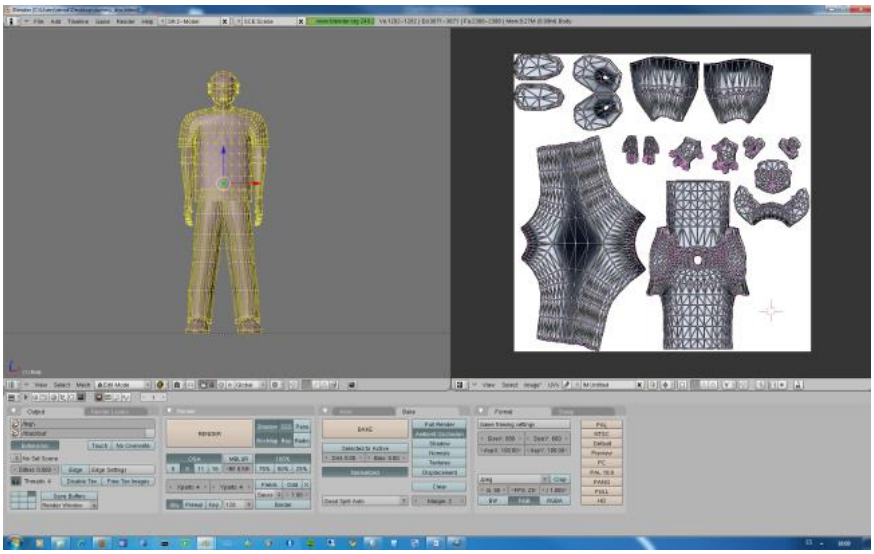

Imagen 68

Una vez calculadas las sombras obtenemos la siguiente imagen (Imagen 69):

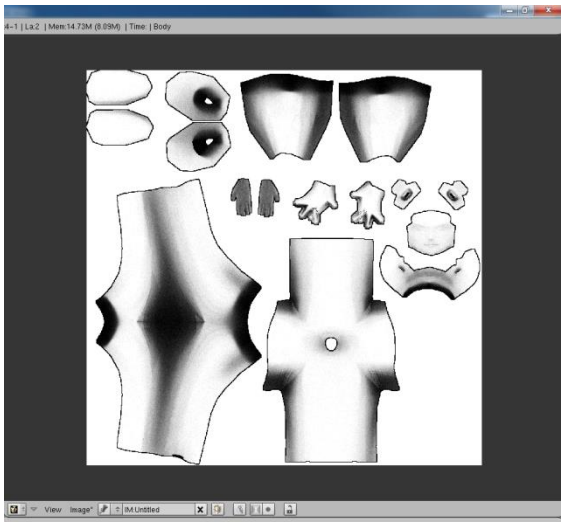

Imagen 69

Exportamos la imagen al formato TGA seleccionando "Image" -> "Save As ...", para poder cargarla con GIMP (Imagen 70):

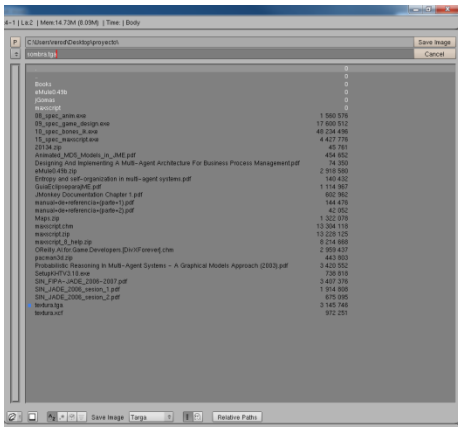

Imagen 70

Una vez guardada la imagen, podemos eliminar el efecto "Subsurf" del modelo ya que no queremos cargar el simulador con tantos polígonos.

Ahora, abrimos con GIMP la textura que hemos coloreado antes, y a continuación, seleccionamos "Archivo" -> "Abrir como capas ..." (Imagen 71):

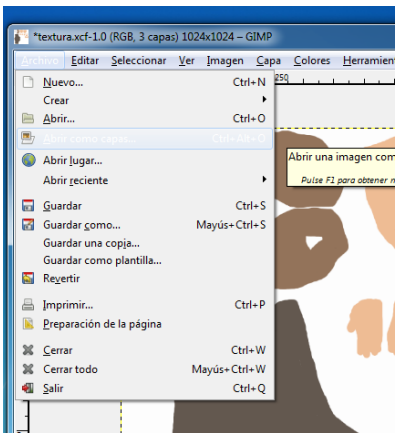

Imagen 71

Y seleccionamos el fichero con las sombras que acabamos de exportar y la colocamos debajo de la capa coloreada (Imagen 72):

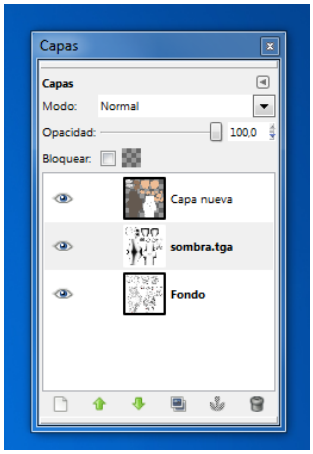

Imagen 72

Pero aún en esta imagen no se aprecian las sombras (Imagen 73):

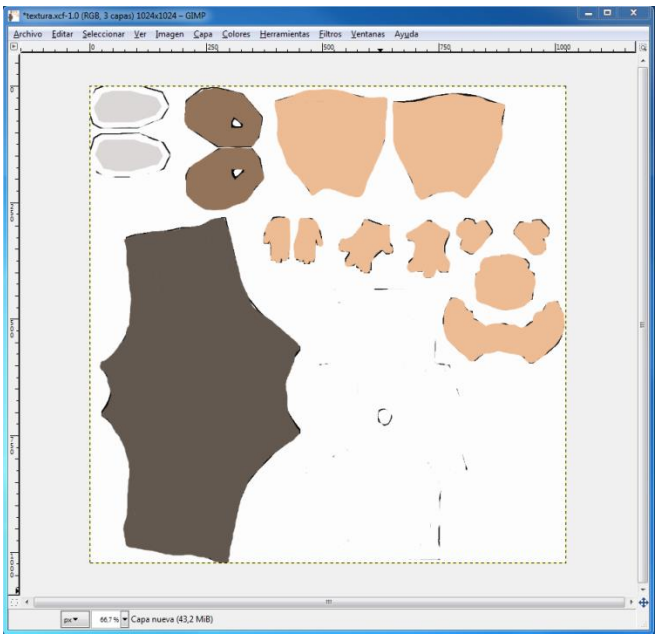

Imagen 73

Para mezclar las sombras con los colores bajamos la "Opacidad" de la primera capa para que se puedan ver las sombras de la capa inferior (Imagen 74):

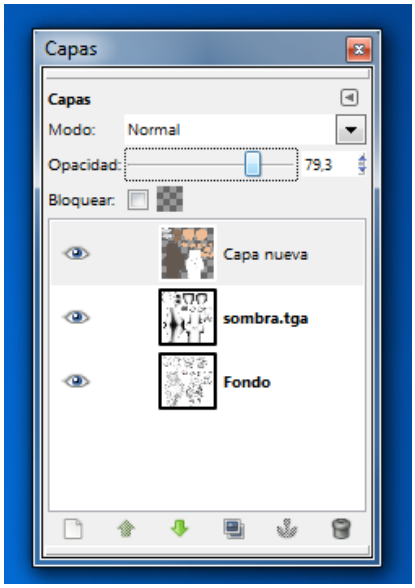

Imagen 74

Una vez hecho esto, obtenemos el siguiente resultado con las sombras dibujadas (Imagen 75):

Imagen 75

Ahora que tenemos la textura con sombras, ya podemos exportarla a un fichero de imagen que soporte nuestro simulador, en este caso lo exportamos a JPG.

En este punto, se termina la creación de la textura, sin embargo, se pueden añadir más detalles utilizando las capas y la opacidad para mezclar texturas a partir de una foto o un dibujo. Por ejemplo, para que los pantalones parezcan pantalones vaqueros, podemos sacar una foto a nuestros vaqueros y añadir una nueva capa encima de los pantalones y mezclarlas regulando la opacidad. Podemos hacer lo mismo con la piel y con el resto de las superficies.

Este es nuestro modelo con la textura cargada (Imagen 76):

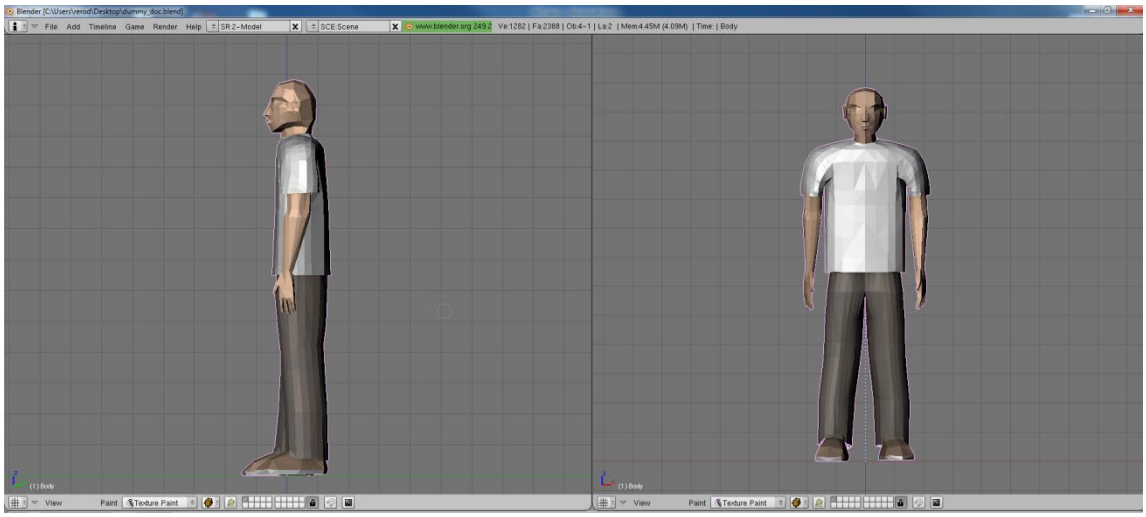

Imagen 76

A2.4 CREACIÓN DE LA ARMATURE Y RIGGING

La *armature* es el esqueleto del modelo. Para dar movimiento a nuestra malla, la forma más utilizada y cómoda en la actualidad es mediante *rigging*. Con esta técnica, sólo tenemos que mover alguna parte de la *armature* y los polígonos de la malla se moverán junto a ella. La función de la *armature* es muy similar a la función que hace el esqueleto de cualquier animal.

A2.4.1 CREACIÓN DE LA ARMATURE

Vamos a empezar por la columna vertebral, abrimos dos "3D Views" uno con el avatar mirando de frente y el otro de lado (Imagen 77):

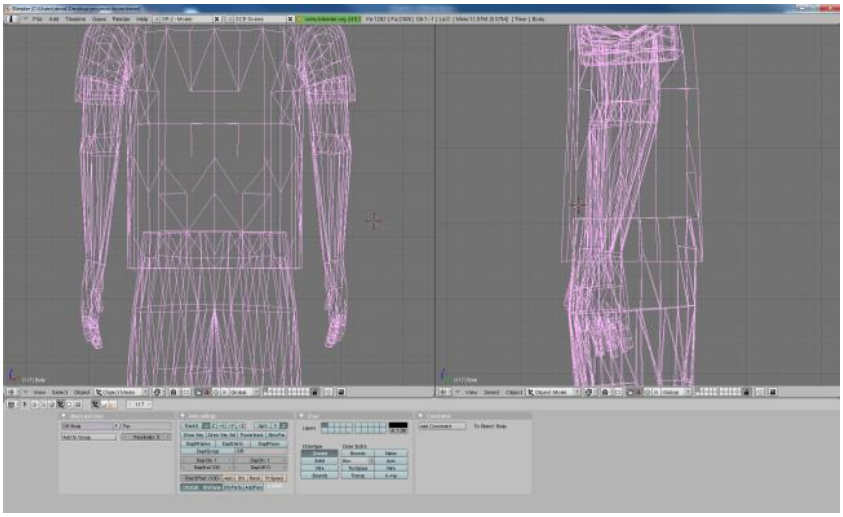

Imagen 77

Colocamos el Cursor justo en el centro del cuerpo a la altura del ombligo. Para ello colocamos el cursor en el centro, primero desde la vista frontal (Imagen 78) y después desde la vista lateral (Imagen 79):

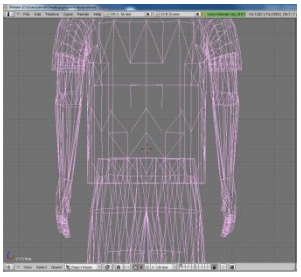

Imagen 78

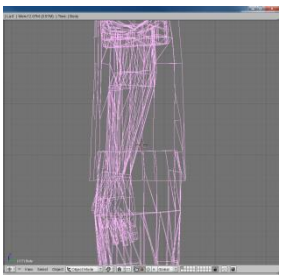

Imagen 79

Ahora, los alineamos con la rejilla del fondo para que quede bien centrado. Si no centramos bien el cursor en este punto podemos tener problemas para crear simétricamente los huesos ya que el eje de simetría estará desplazado con respecto al eje de la malla.

Pues bien, pulsamos "Shift+S" al lado del cursor y seleccionamos "Cursor -> Grid" (Imagen 80):

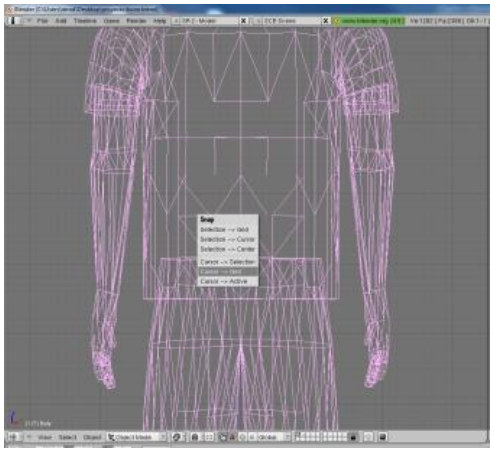

Imagen 80

Con esto ya tenemos el cursor alineado a la cuadrícula más cercana, tanto en la vista frontal como lateral. Ahora pulsamos "Espacio" y seleccionamos "Add" -> "Armature" para crear el primer hueso (Imagen 81):

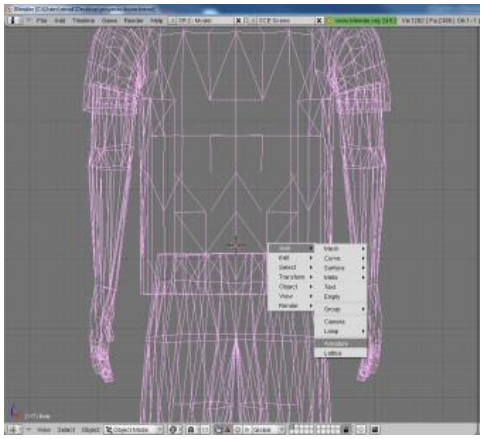

Imagen 81

Ya tenemos el primer hueso colocado en el centro del cuerpo (Imagen 82):

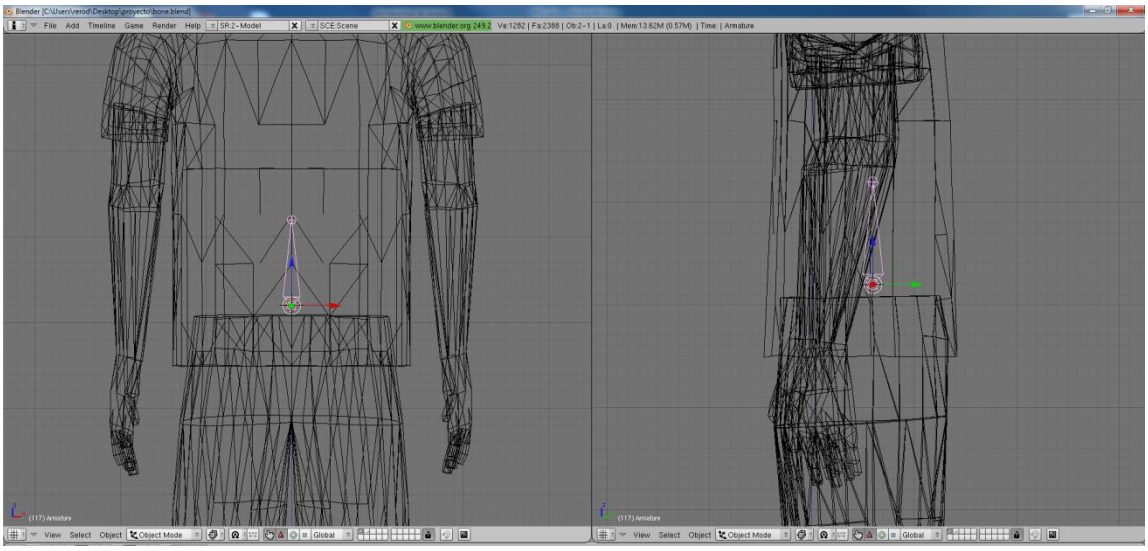

Imagen 82

Pulsamos "Tabulador" para entrar en modo de edición y seleccionamos uno de los extremos del hueso (Imagen 83):

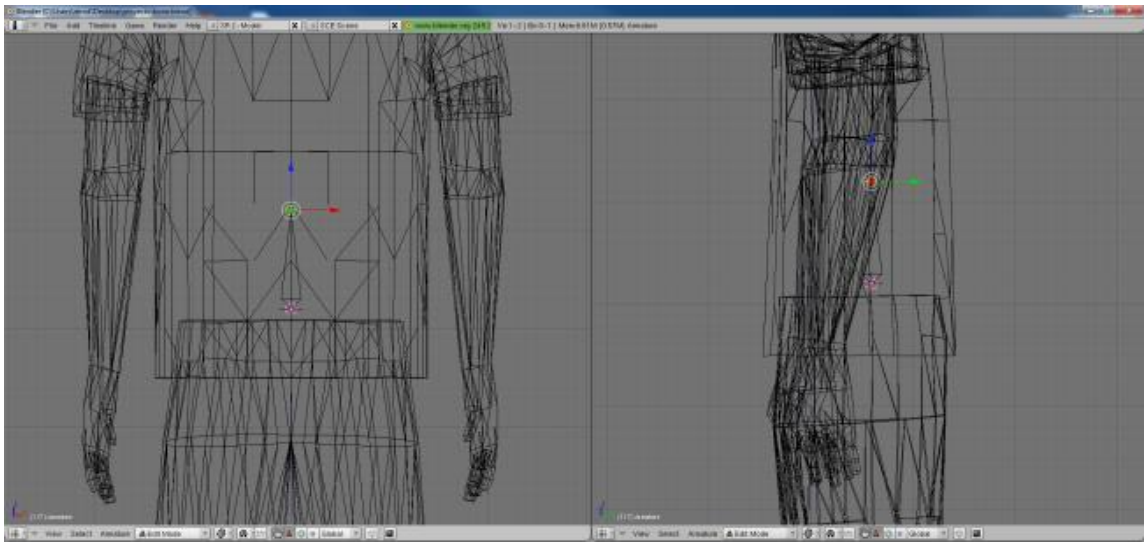

Imagen 83

Primero, creamos la columna. Pulsamos "E" para crear otro hueso conectado al extremo seleccionado (Imagen 84):

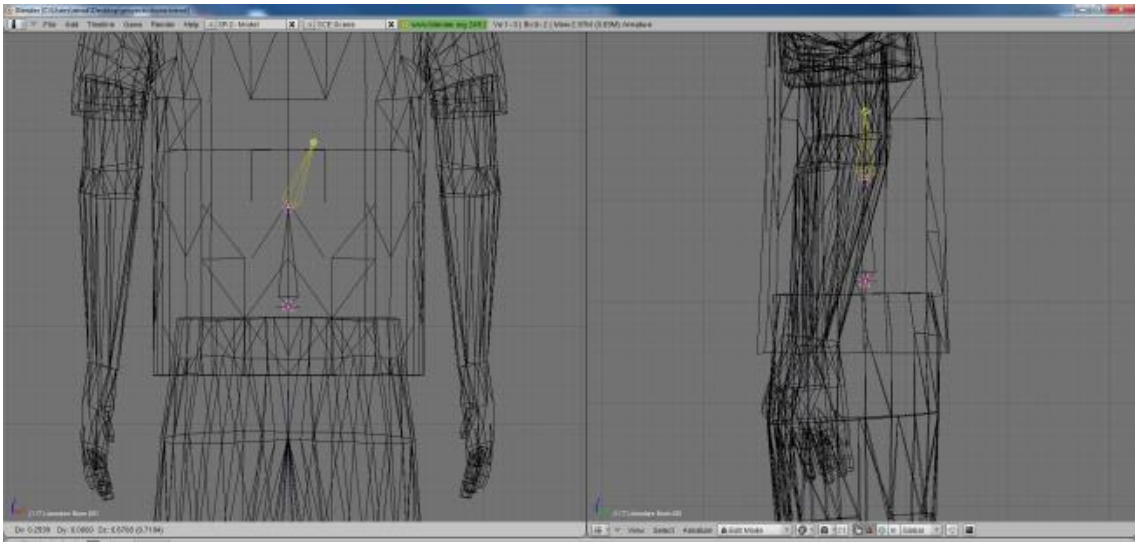

Imagen 84

Para que no se tuerza la columna pulsamos "Z" para indicar que lo queremos mover sólo sobre el eje Z (Imagen 85):

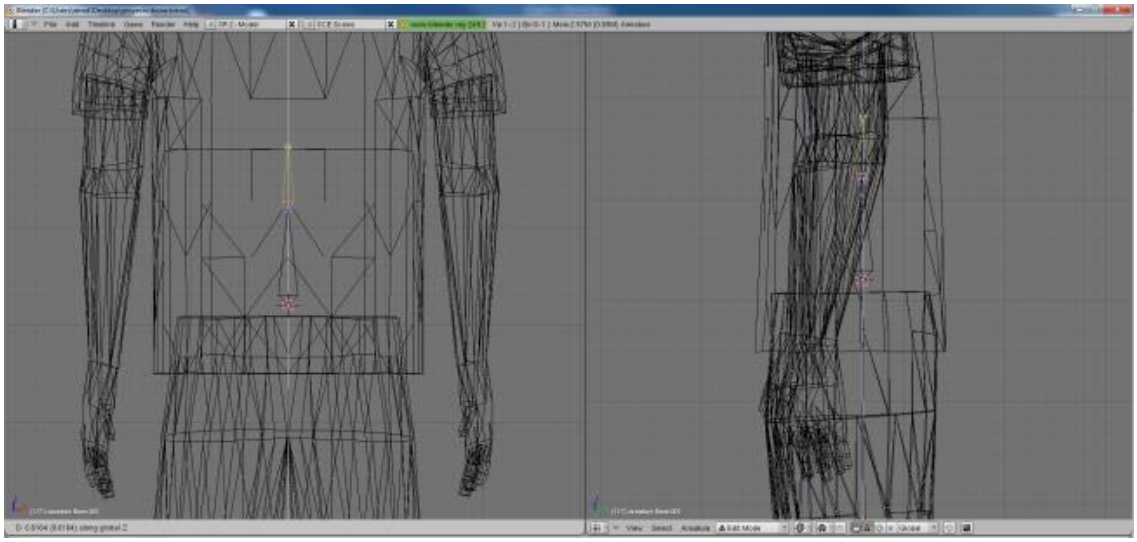

Imagen 85

Una vez que tenemos la longitud deseada hacemos clic izquierdo para materializar el hueso.

Seleccionamos el extremo del hueso y pulsamos "E" para crear otro hueso, y repetimos la operación hasta llegar al cuello (Imagen 86):

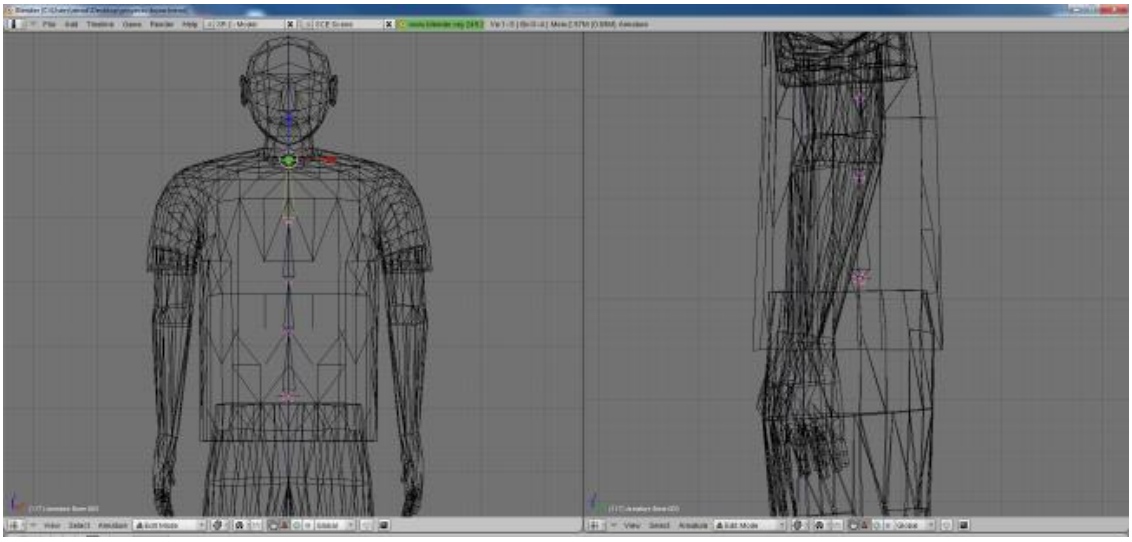

Imagen 86

Ahora creamos los huesos de los hombros. Como la malla del avatar el simétrica, podemos crear los huesos de forma simétrica. Para ello, debemos tener seleccionado el botón "X-Axis Mirror" en la pestaña "Armature" (Imagen 87):

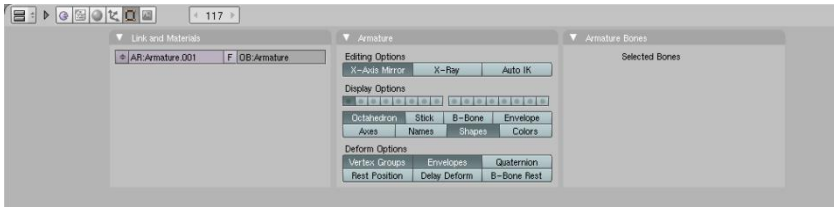

Imagen 87

Ahora seleccionamos el extremo del hueso que se encuentra en el cuello y pulsamos "Shift+E" para que nos cree dos huesos nuevos, uno en cada lado (Imagen 88):

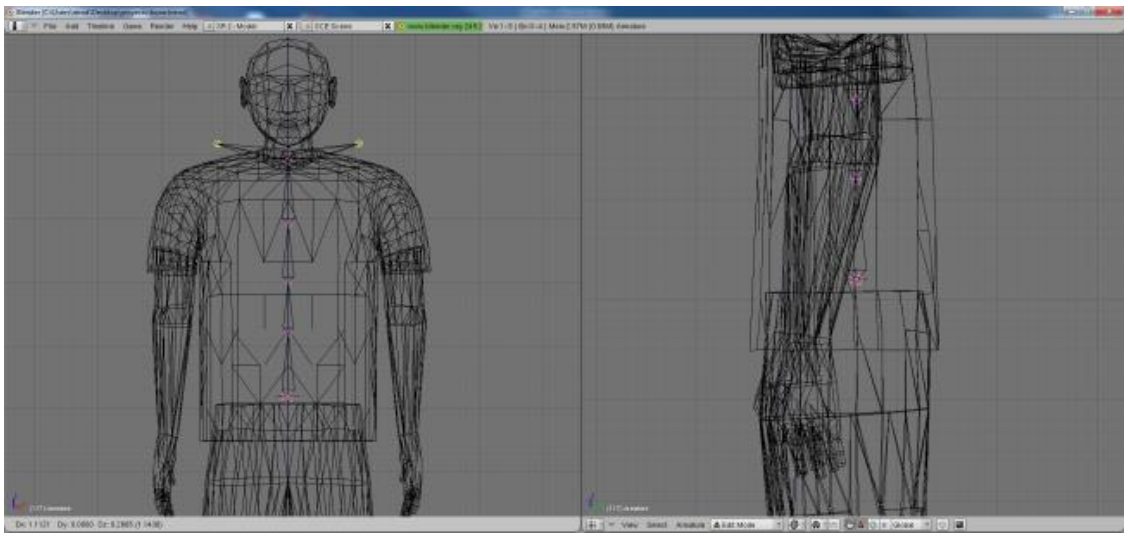

Imagen 88

De esta forma, con sólo manipular uno de los dos huesos, el otro hueso se coloca de forma simétrica en el lado opuesto.

Ahora creamos los huesos de los brazos, en este caso, seleccionamos uno de los huesos de los hombros y pulsamos "E" para crear el primer hueso del brazo. Esto hace que se cree otro hueso exactamente igual en el otro hombro (Imagen 89):

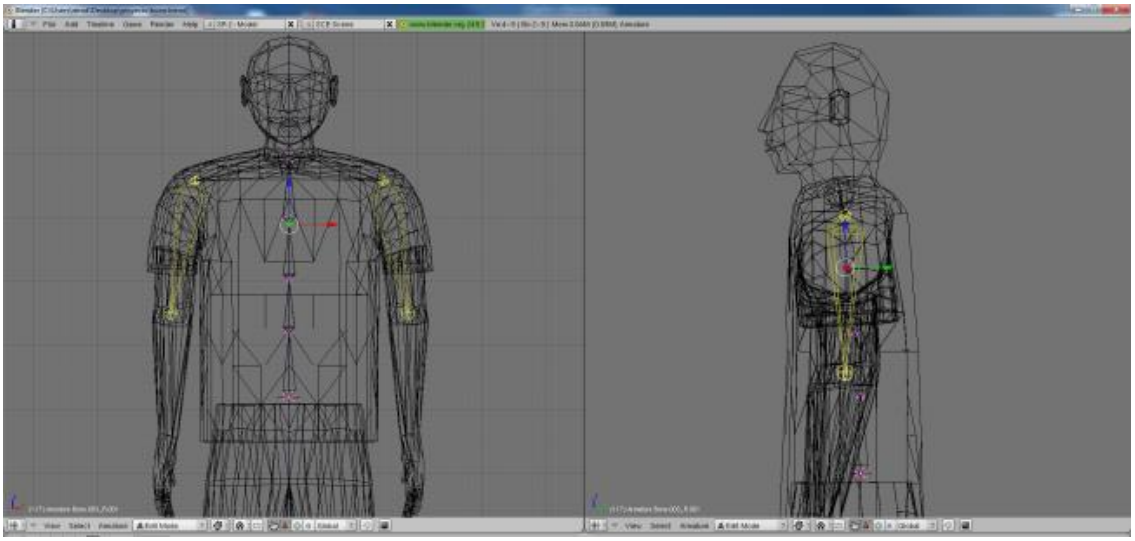

Imagen 89

Terminamos los huesos de la cintura para arriba, y nos quedará lo siguiente (Imagen 90):

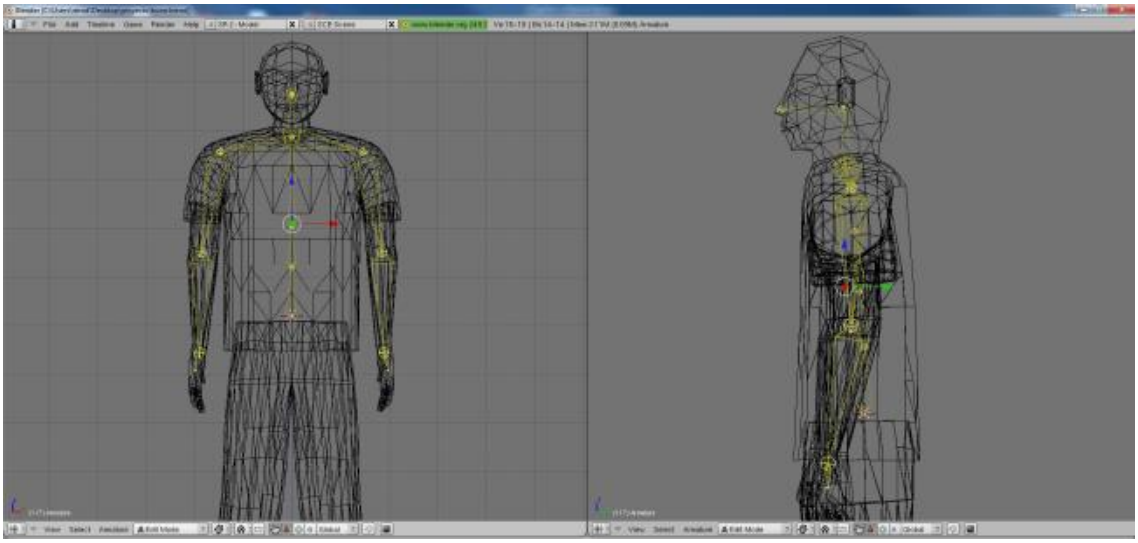

Imagen 90

A continuación, pasamos a los huesos de la cintura para abajo. Para empezar, seleccionamos el extremo libre del primer hueso que hemos creado (Imagen 91):

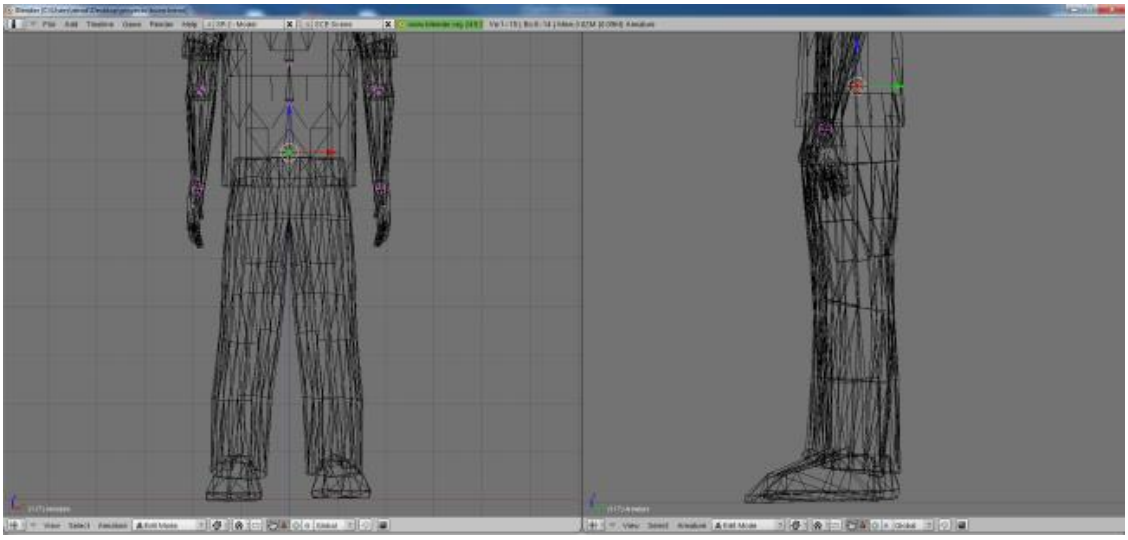

Imagen 91

Creamos dos huesos simétricos que irán conectados con los huesos de las piernas. Para ello, pulsamos "Shift+E" y colocamos los huesos nuevos en la posición deseada (Imagen 92):

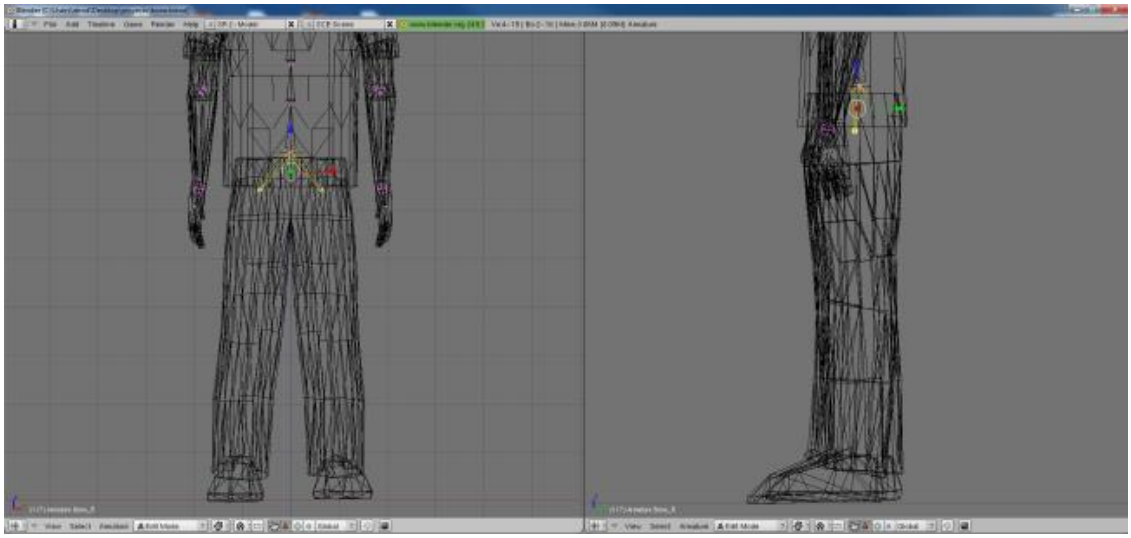

Imagen 92

Seleccionamos el extremo libre de uno de los dos huesos que acabamos de crear y pulsamos "E" para crear el primer hueso de la pierna y completamos el resto de los huesos (Imagen 93):

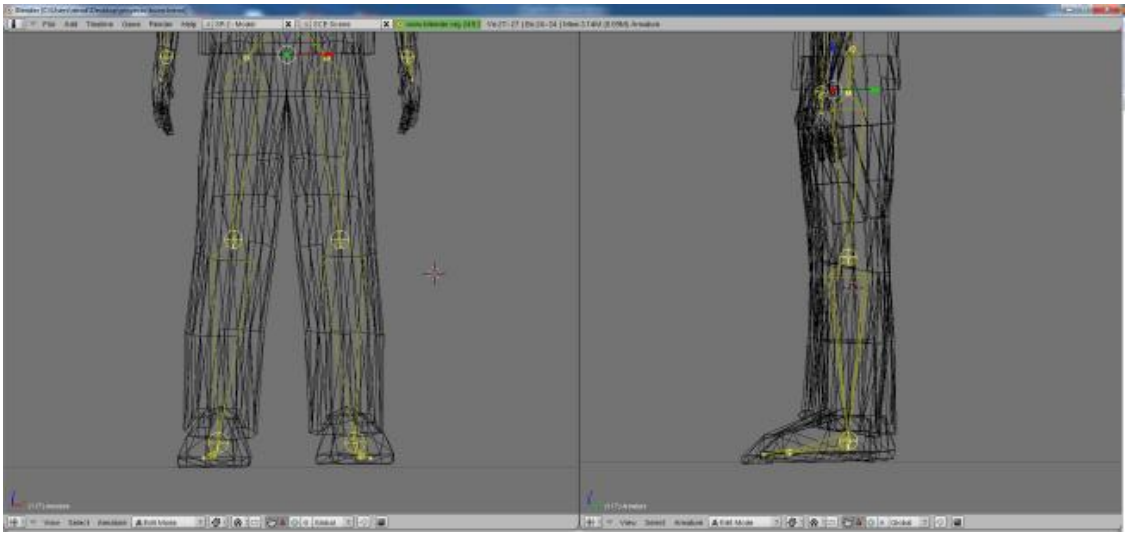

Imagen 93

Con esto tenemos el esqueleto completo (Imagen 94):

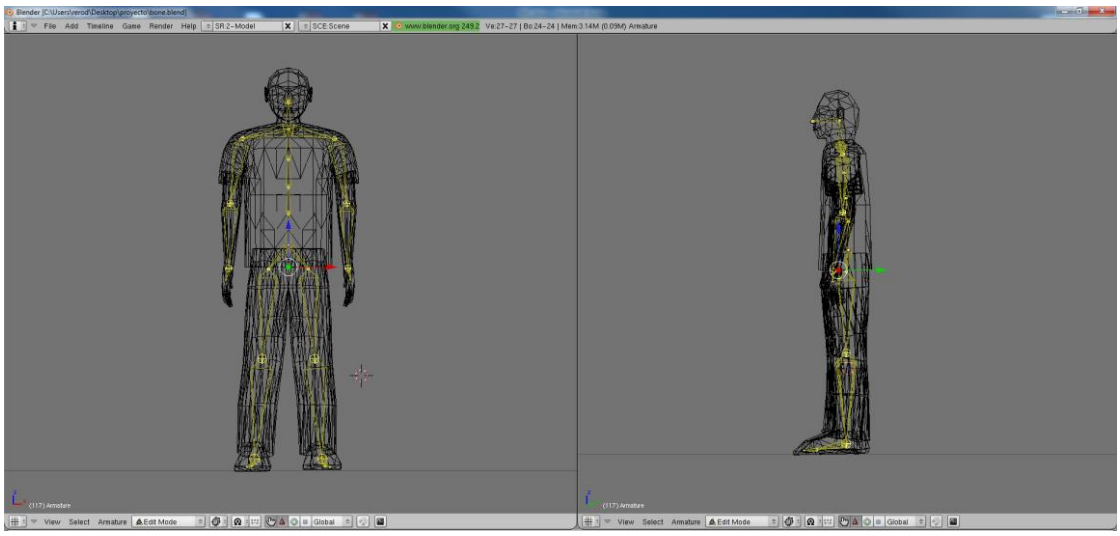

Imagen 94

Una vez hecho el esqueleto, vamos a añadir algunas restricciones en algunos huesos para que nos sea más fácil crear las poses del avatar.

Para empezar, salimos del modo edición de los huesos pulsando "Tabulador". Ahora seleccionamos el esqueleto y pulsamos "Ctrl+Tabulador" para entrar en modo pose. En este modo podemos manipular los huesos para crear las distintas poses de la animación (Imagen 95):

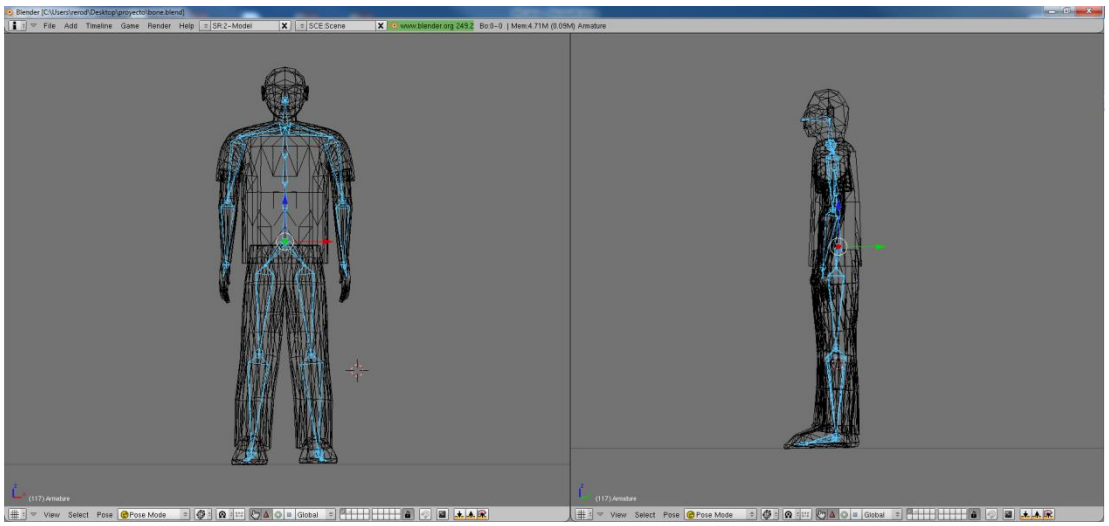

Imagen 95

Vamos a añadir un par de "IK Solvers", una en cada pierna, para que el movimiento de las piernas parezca más natural. Pero para añadir un "IK Solver" necesitamos un hueso extra que haga de manipulador. Por eso, vamos a añadir dos huesos más en las piernas, uno en cada pierna.

Para añadir los huesos nuevos, tenemos que volver al modo edición de huesos. Pulsamos "Ctrl+Tabulador" para salir del modo pose y seleccionamos el esqueleto y pulsamos "Tabulador" para poder editar los huesos.

Empezamos por la pierna izquierda. Seleccionamos el talón del pie izquierdo (Imagen 96):

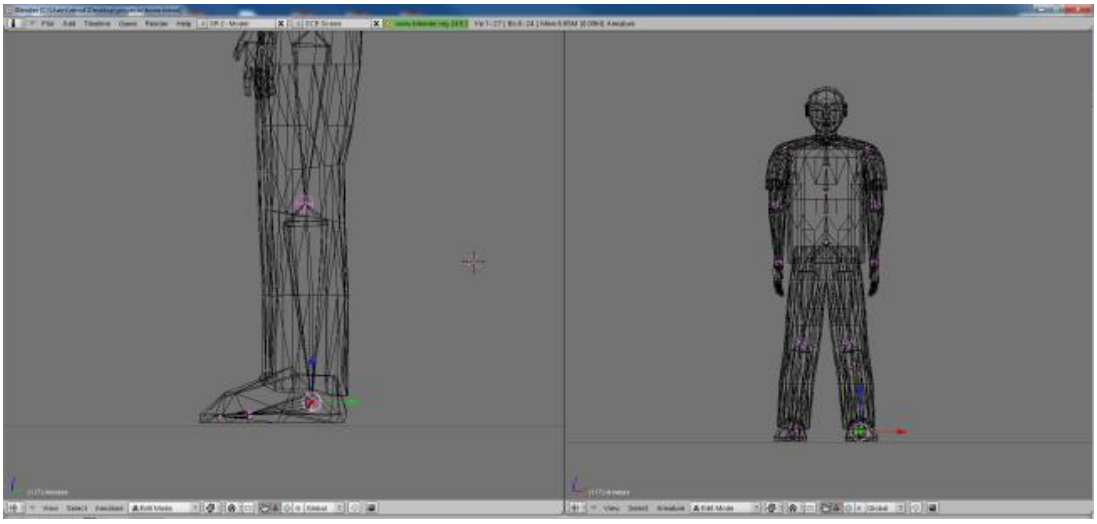

Imagen 96

Pulsamos "Shift+S" y elegimos "Cursor -> Selection" para colocar el cursor justo encima de la selección (Imagen 97):

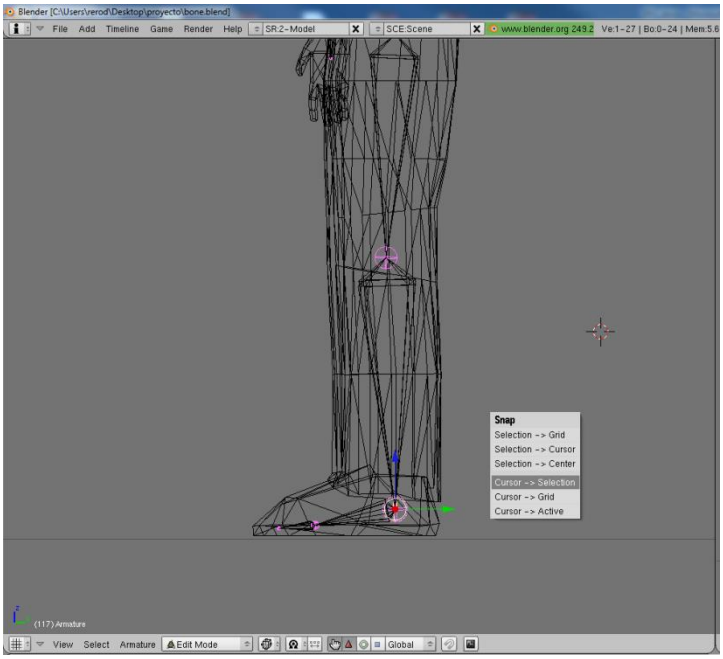

Imagen 97

Una vez colocado el cursor en el extremo del hueso creamos un hueso en esa posición pulsando "Espacio" y seleccionando "Add" -> "Bone" y colocamos el hueso nuevo en el talón (Imagen 98):

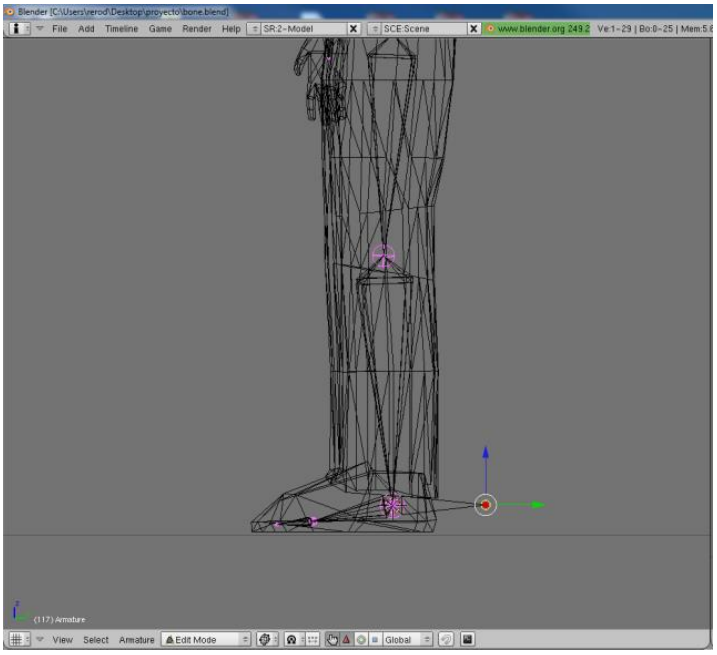

Imagen 98

Este hueso no debe estar conectado a ningún otro hueso, ya que no forma parte del esqueleto. Se trata solamente de un manipulador que vamos a manejar nosotros más adelante.

Ahora creamos el hueso de la pierna derecha siguiendo exactamente los mismos pasos y obtenemos el siguiente resultado (Imagen 99):

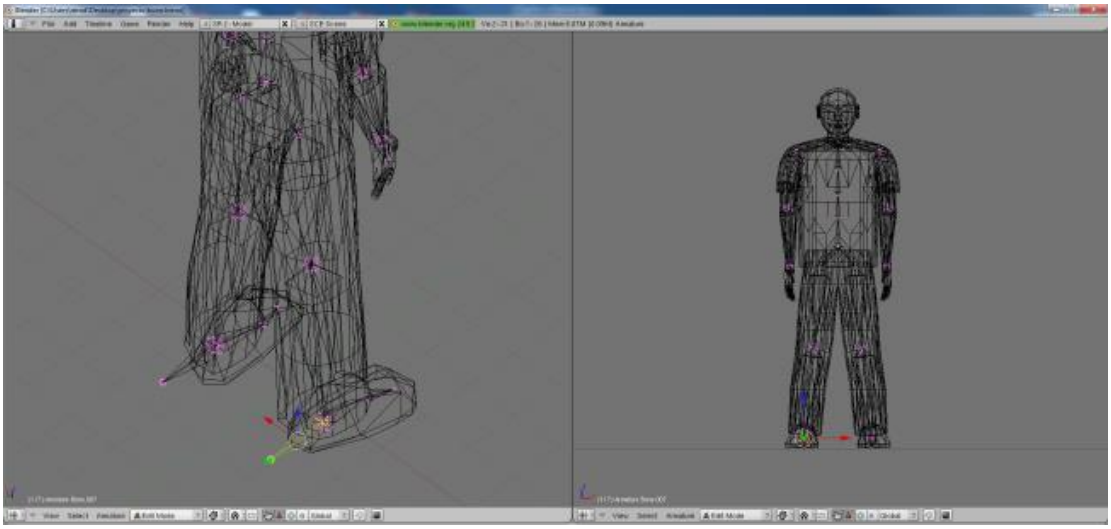

Imagen 99

Ya podemos volver al modo pose para añadir los "IK Solvers". Pulsamos "Tabulador", seleccionamos el esqueleto y pulsamos "Ctrl+Tabulador", y seleccionamos el primer hueso al que vamos a añadir el "IK Solver" (Imagen 100):

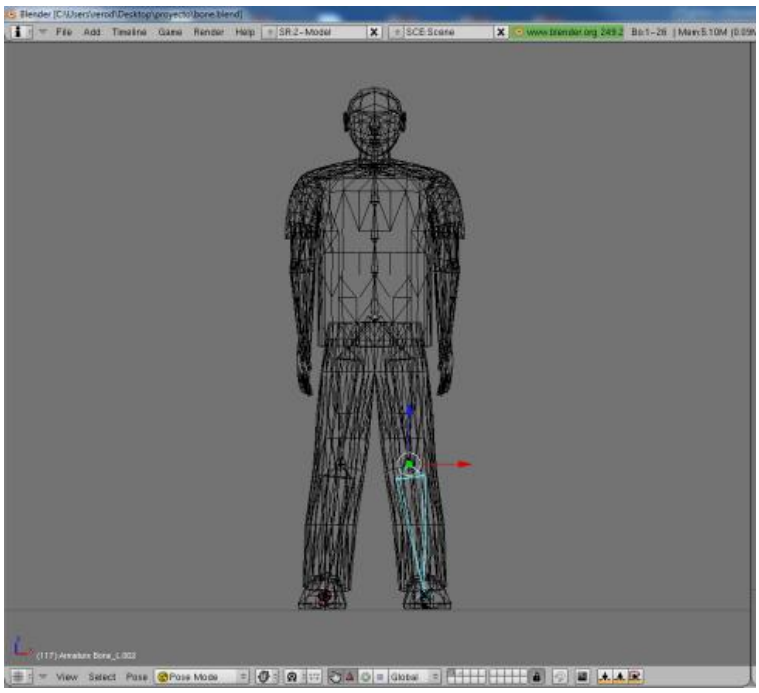

Imagen 100

Una vez seleccionado el hueso pulsamos el botón "Add Constraint" que se encuentra en el área inferior, justo dentro de la pestaña "Constraints". Y en el menú desplegado seleccionamos "IK Solver" (Imagen 101):

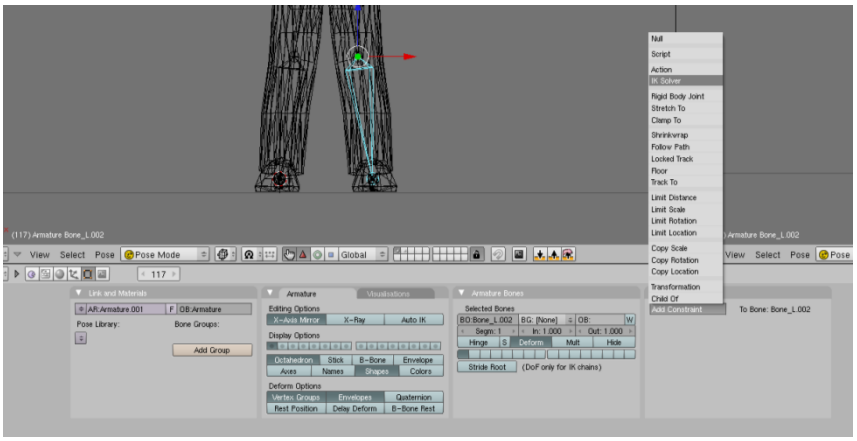

Imagen 101

Y nos aparecerá el siguiente recuadro (Imagen 102):

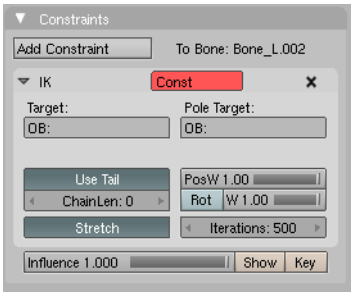

Imagen 102

En la casilla "Target:" introducimos el nombre del esqueleto, en este caso "Armature", y pulsamos "Intro" y aparecerá otra casilla en la que pone "BO:". Aquí introducimos el nombre del hueso que hemos creado en el talón de este pie, en este caso "Bone.006" (Imagen 103):

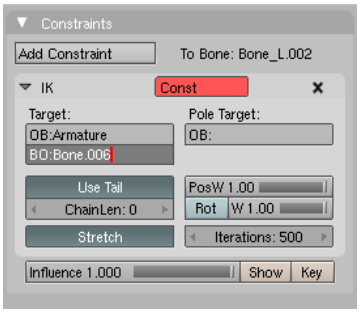

Imagen 103

En este estado, los movimientos del hueso seleccionado se propagarán por todos los huesos conectados (Imagen 104). Pero nuestro propósito es que se propague hasta la cadera, para ello modificamos el campo "ChainLen:". Este campo indica el número de huesos adyacentes a los que permite propagar el movimiento del hueso que nos sirve de manipulador.

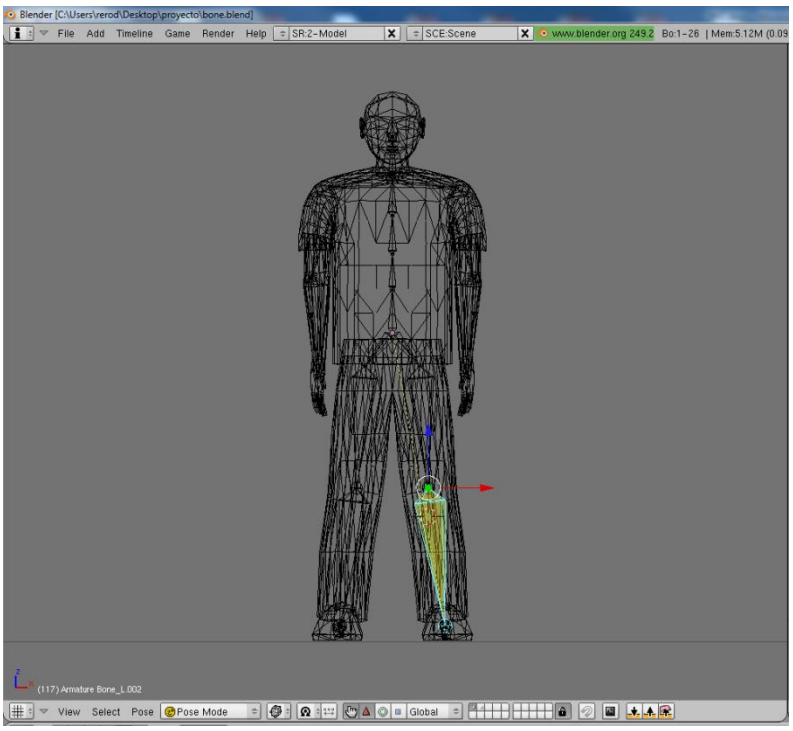

Imagen 104

Para nuestro ejemplo, tenemos dos huesos desde el manipulador hasta llegar a la cadera por lo que debemos poner "ChainLen: 2" (Imagen 105):

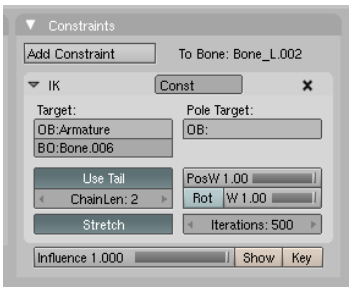

Imagen 105

Y observamos que la línea discontinua, que antes estaba conectada con el centro del cuerpo, ahora sólo llega hasta la cadera (Imagen 106):

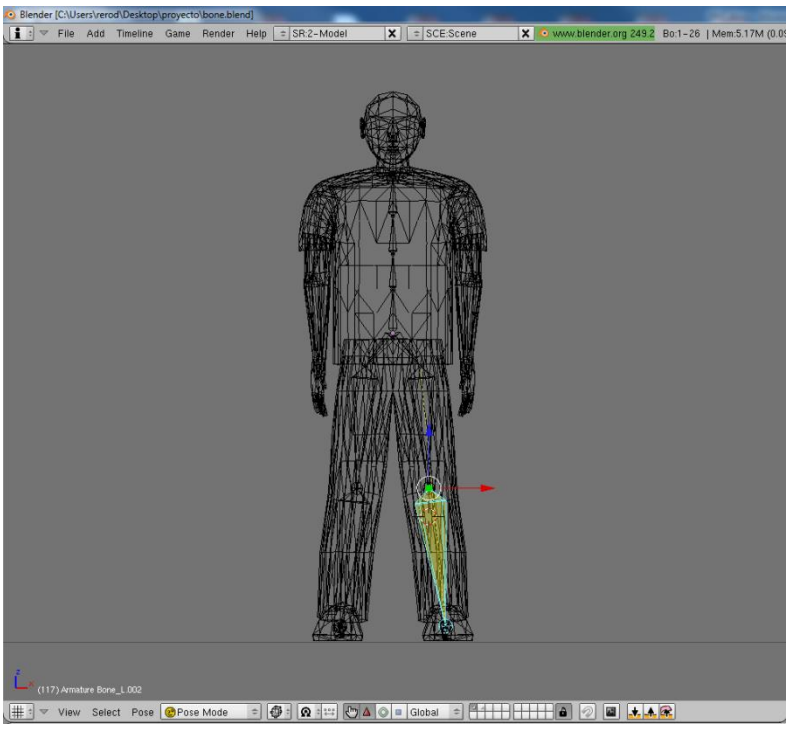

Imagen 106

Si movemos el hueso "manipulador", podemos observar que la pierna izquierda sigue su trayectoria flexionando la rodilla (Imagen 107, Imagen 108 e Imagen 109):

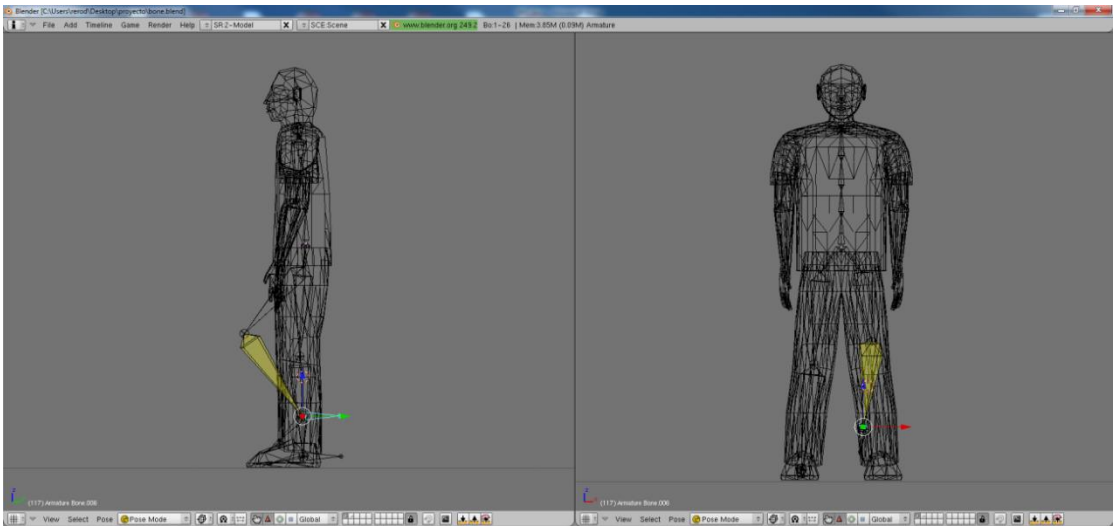

Imagen 107

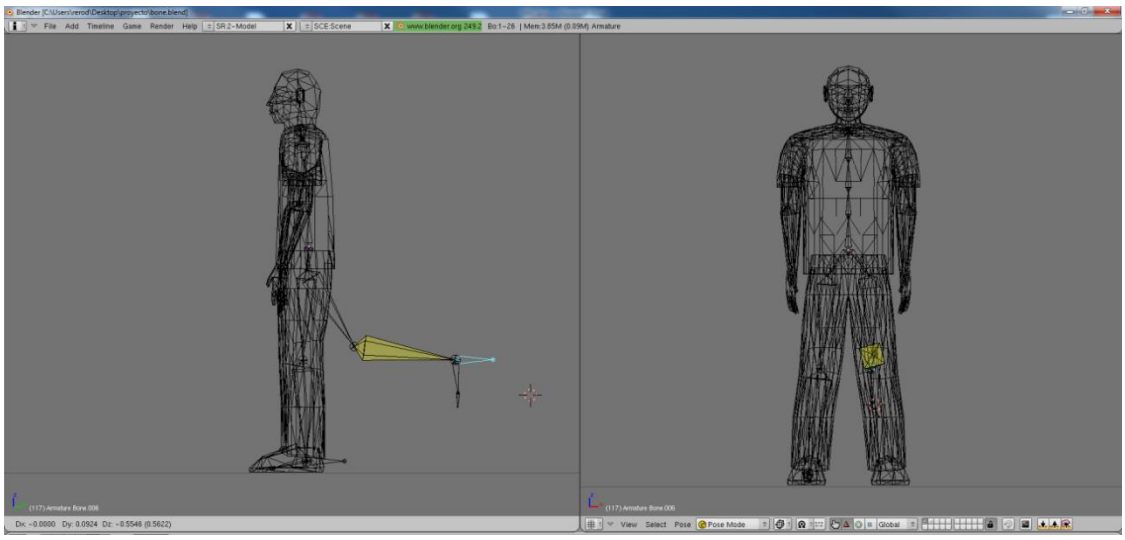

Imagen 108

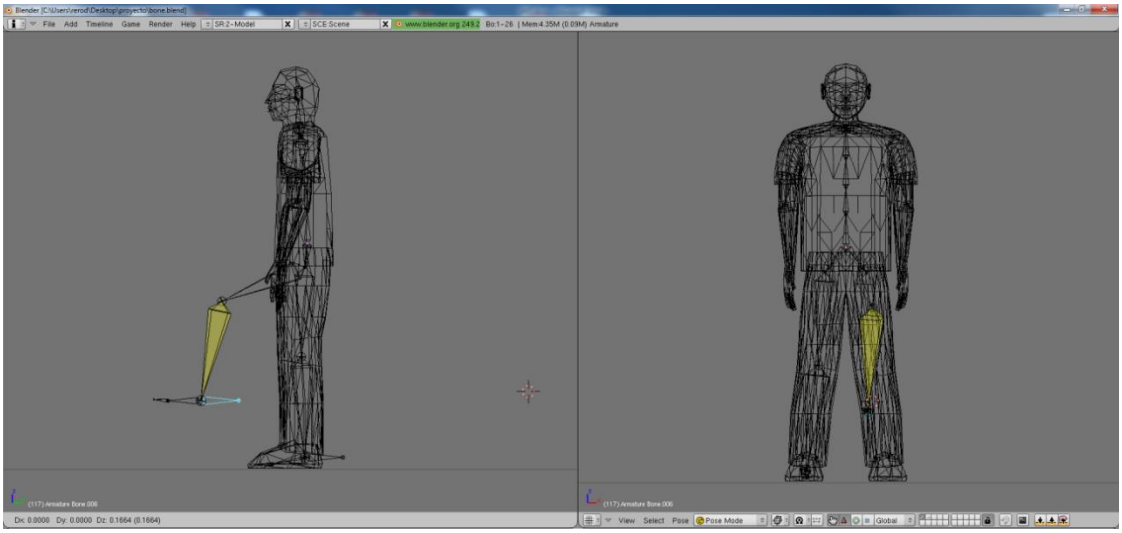

Imagen 109

Repetimos la operación con la otra pierna, y los resultados son los siguientes (Imagen 110 e Imagen 111):

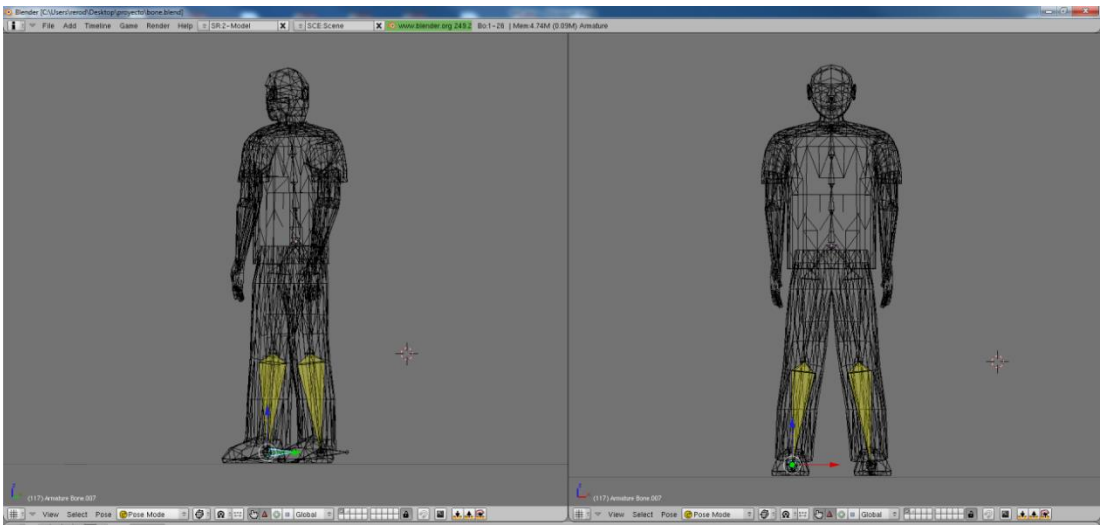

Imagen 110

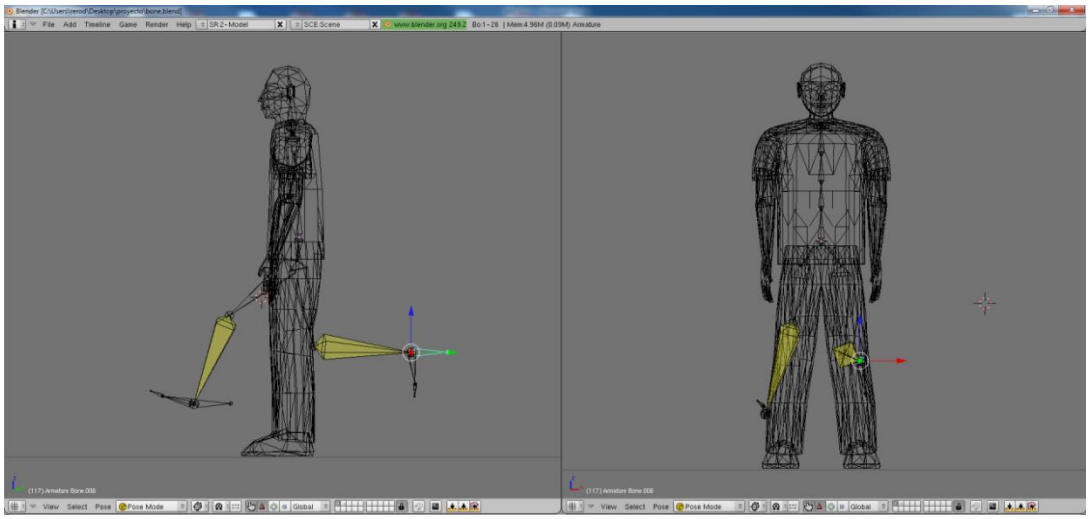

Imagen 111

Ahora vamos a añadir los "IK Solvers" en los brazos. Esta vez vamos a añadir dos "IK Solvers" en cada brazo, uno para cada hueso del brazo. Para ello, añadimos dos huesos "manipuladores" en cada brazo. Uno en la muñeca y otro en el codo (Imagen 112):

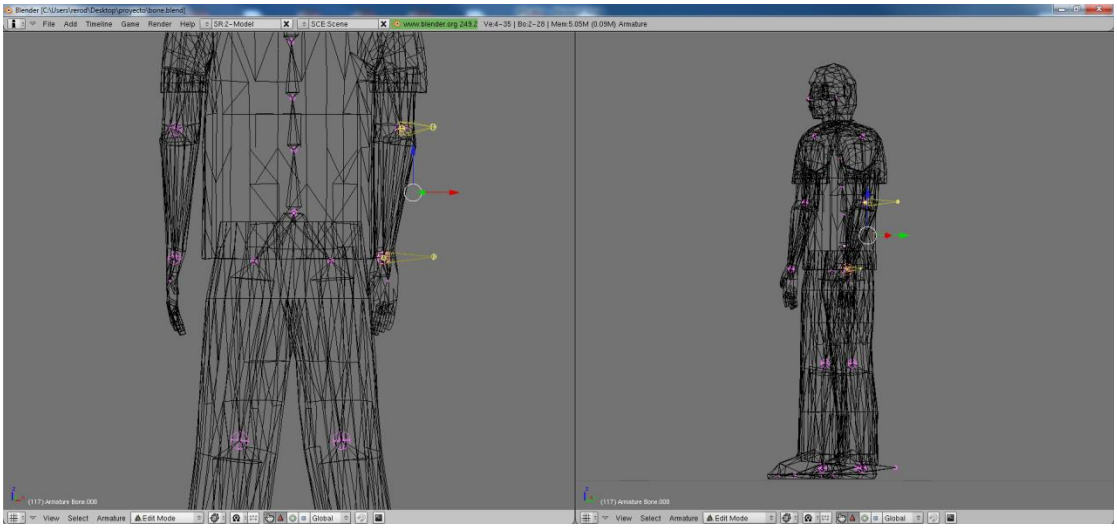

Imagen 112

Y ahora en el otro brazo (Imagen 113):

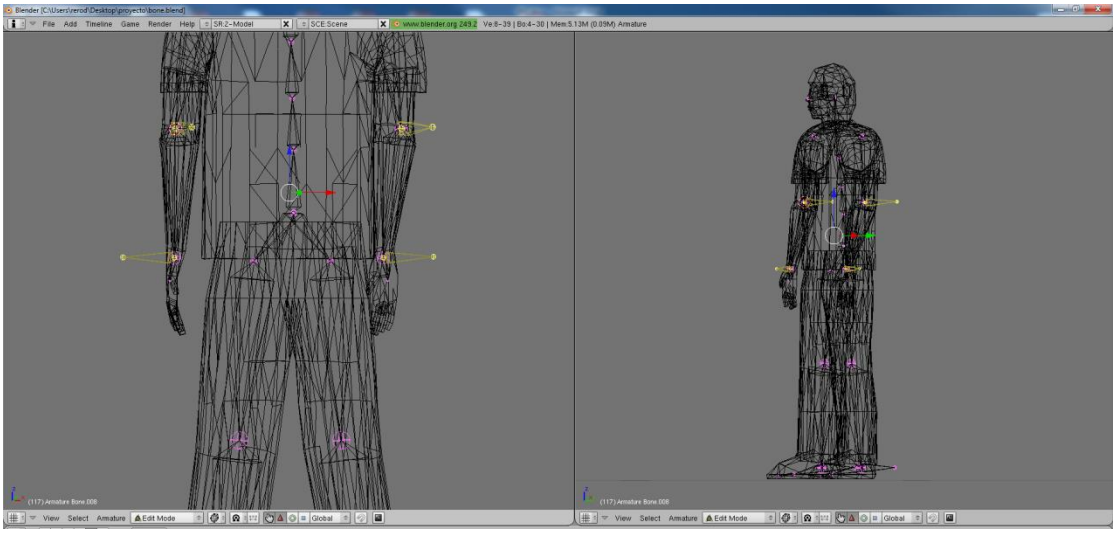

Imagen 113

Una vez hecho esto, añadimos los cuatro "IK Solvers" y nos quedará con el siguiente aspecto (Imagen 114):

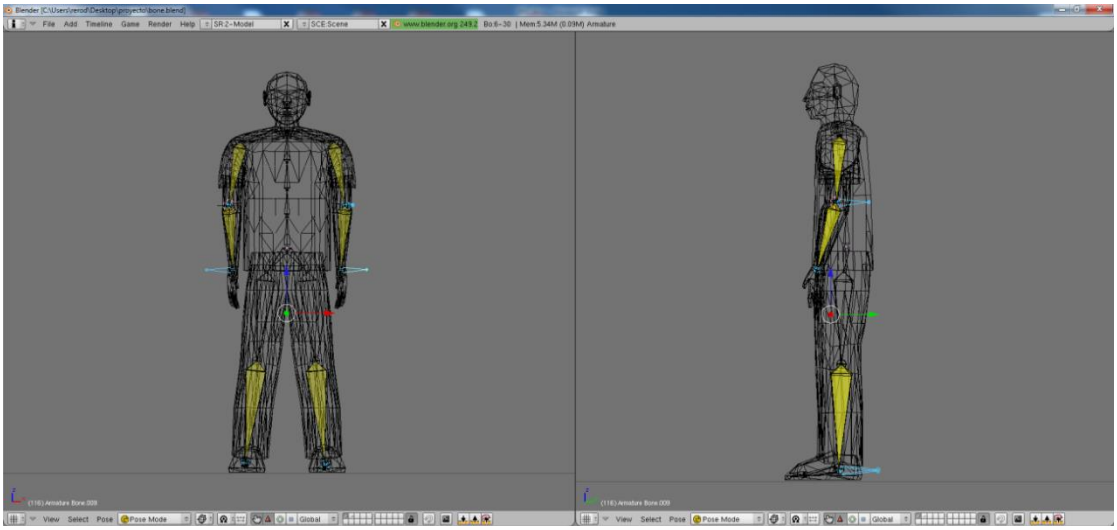

Imagen 114

Ya podemos manipular los brazos también (Imagen 115):

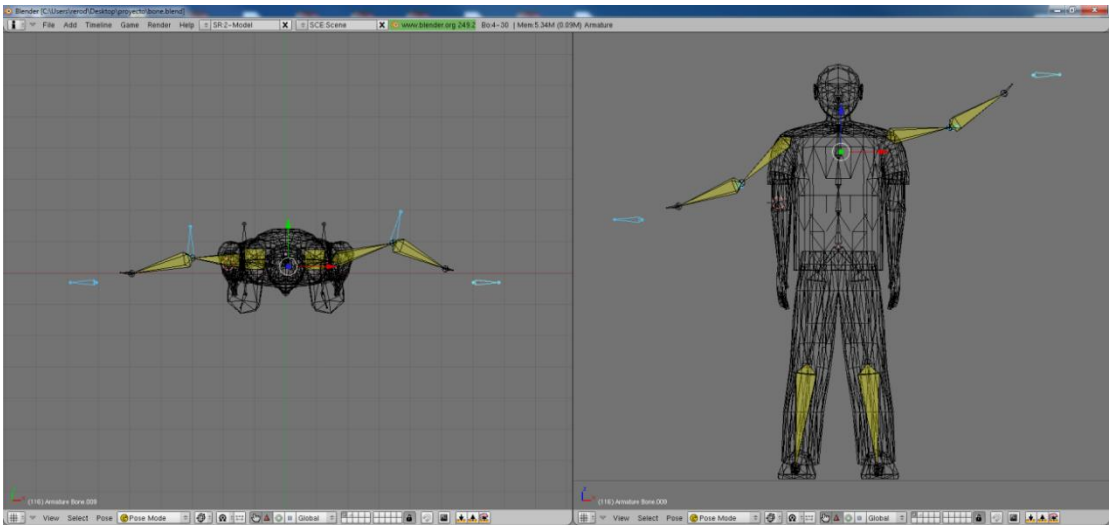

Imagen 115

A2.4.2 RIGGING

Una vez creado el esqueleto, vamos a unirlo con la malla. Esto permitirá deformar la malla según el movimiento de los huesos.

Para empezar, alineamos bien cada uno de los huesos con la malla (Imagen 116):

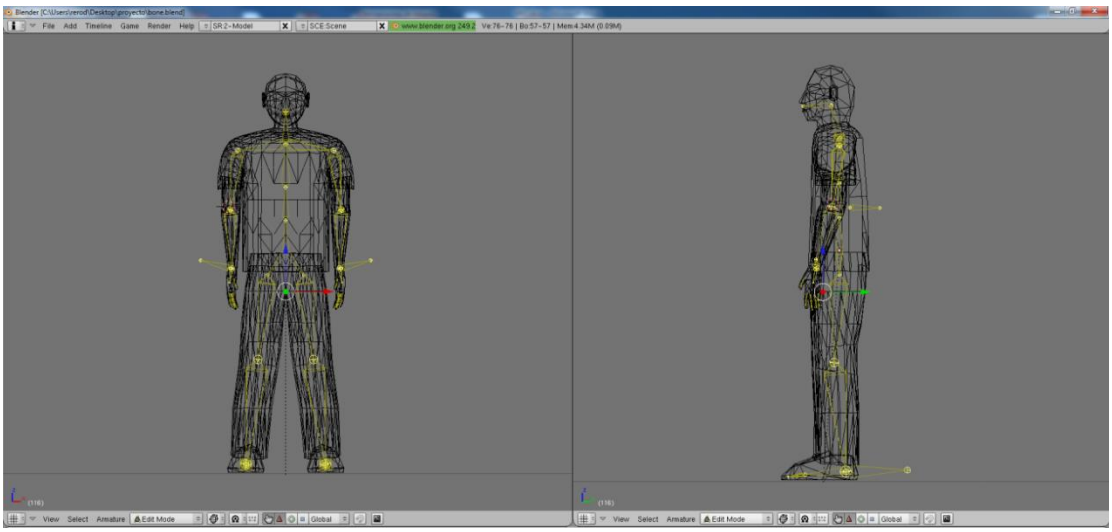

Imagen 116

Como ejemplo, se ha añadido también los huesos de las manos para poder mover los dedos, aunque no es necesario porque no vamos a crear una animación de las manos.

Pues bien, ahora vamos a enlazar la malla con el esqueleto. Para esto, debemos entrar en modo objeto (Imagen 117):

Imagen 117

Y seleccionamos primero la malla (Imagen 118):

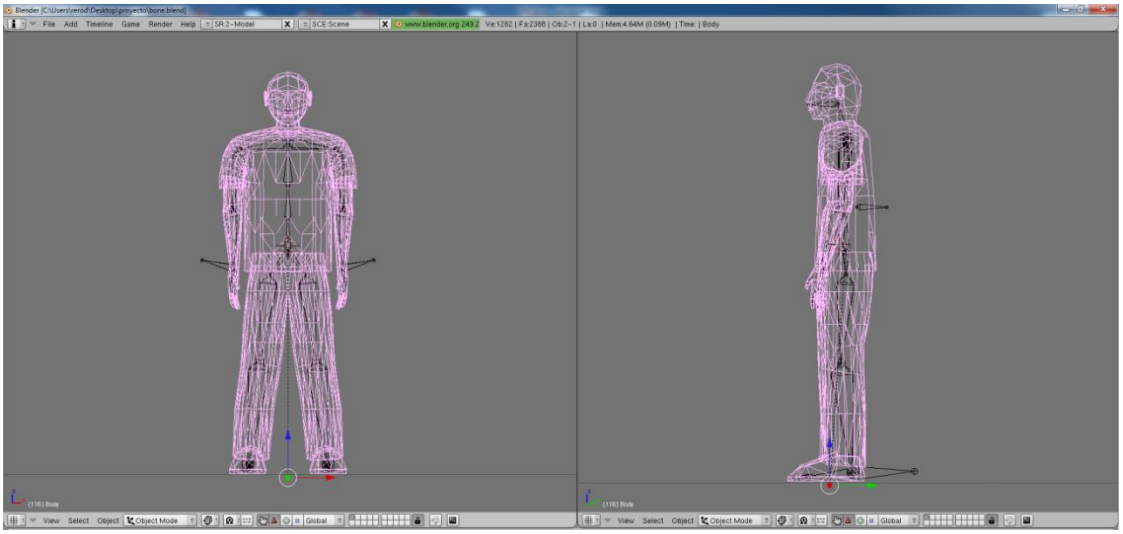

Imagen 118

Pulsamos "Shift" y seleccionamos el esqueleto para que se mantenga seleccionada la malla también (Imagen 119):

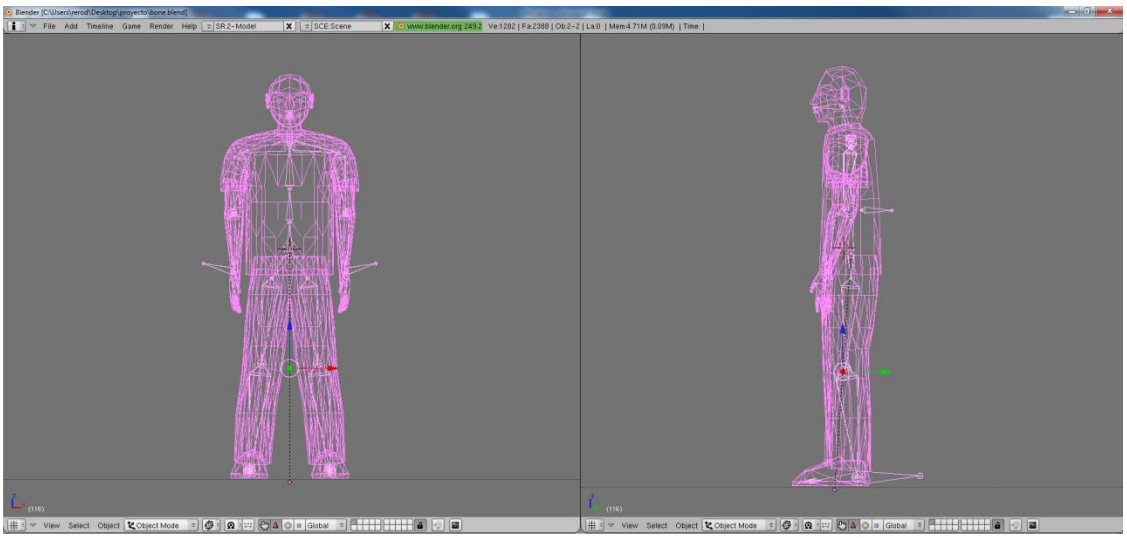

Imagen 119

Para enlazarlos pulsamos "Ctrl+P" y seleccionamos "Armature" (Imagen 120):

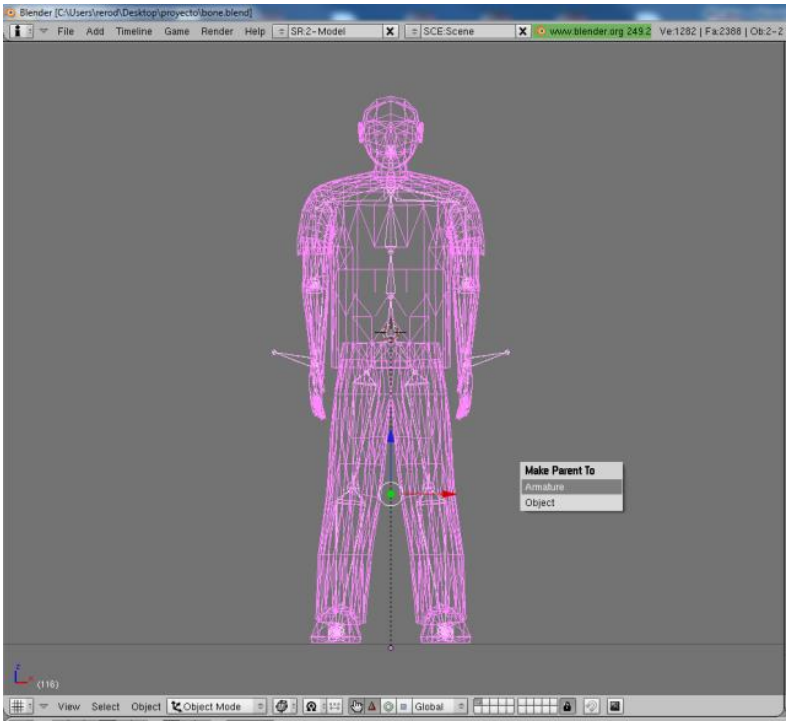

Imagen 120

Y a continuación seleccionamos "Create From Bone Heat" (Imagen 121):

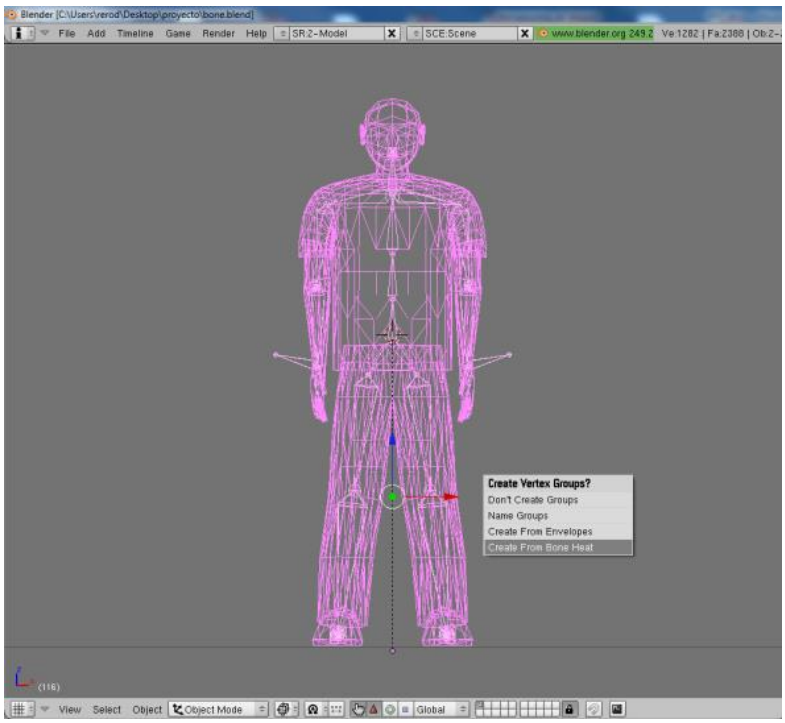

Imagen 121

Esto hace que Blender calcule automáticamente los pesos de influencia según la distancia de los huesos con los vértices de las mallas. Cuanto más cerca esté el vértice de un hueso, éste tendrá mayor influencia a la hora de deformar la malla.

Una vez calculados y asignados los pesos, podemos entrar en modo pose (seleccionar el esqueleto y pulsar "Ctrl+Tabulador") para manipular los huesos, y comprobar que la malla se deforma correctamente (Imagen 122):

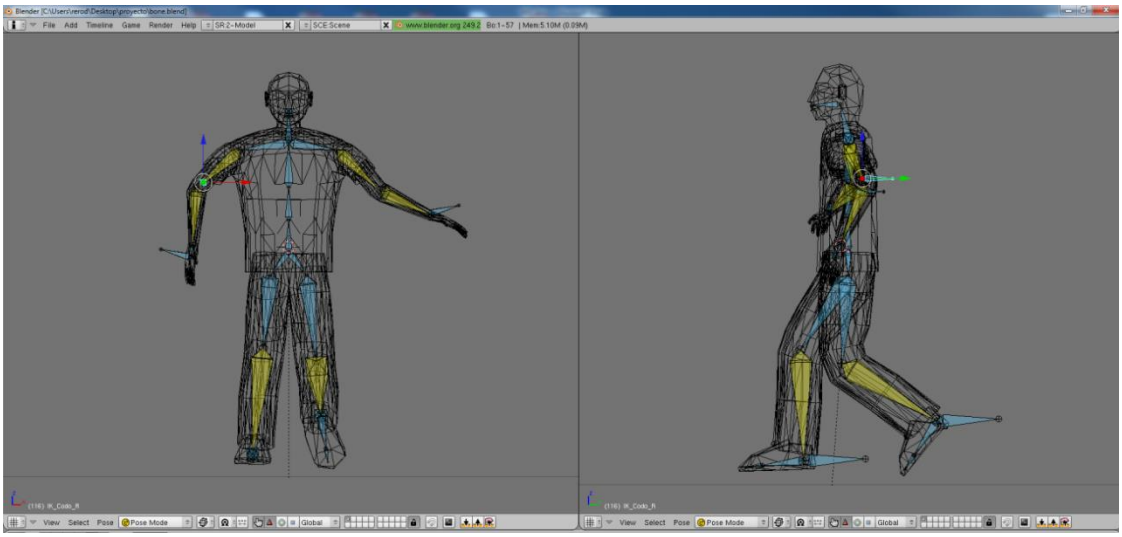

Imagen 122

Movemos bien todas las articulaciones para comprobar que los pesos se han asignado correctamente. Es raro que esto funcione a la primera, siempre hay vértices que realiza un movimiento no deseado.

La siguiente imagen (Imagen 123) muestra uno de esos vértices:

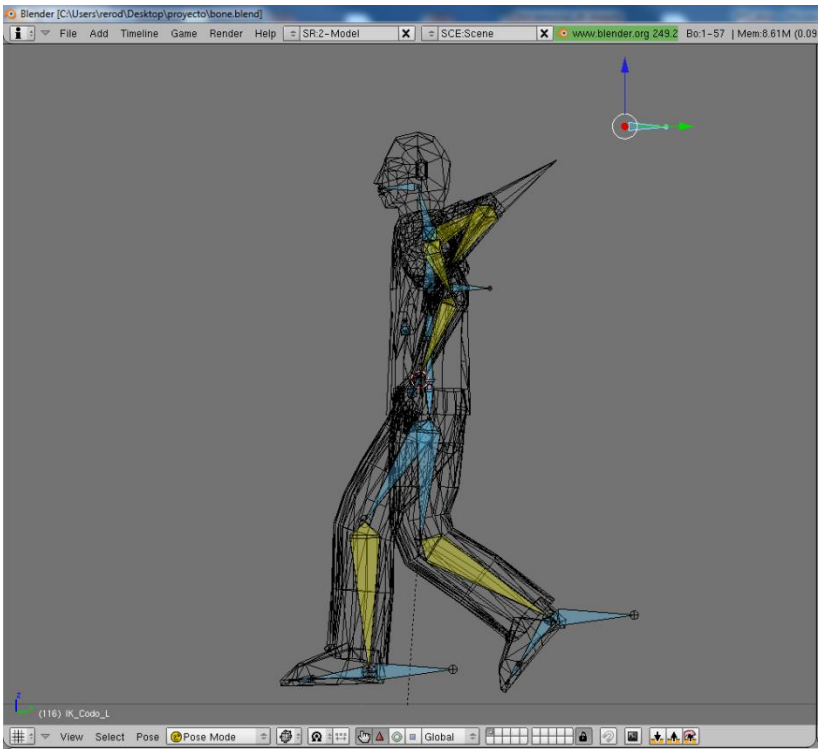

Imagen 123

En esta imagen vemos que uno de los vértices del codo está influenciado por la traslación del hueso "manipulador". Estos defectos los tenemos que solucionar manualmente.

Para corregirlo, tenemos que ir al modo objeto ("object mode") y seleccionar sólo la malla. A continuación, pulamos "Tabulador" para entrar en modo edición (Imagen 124).

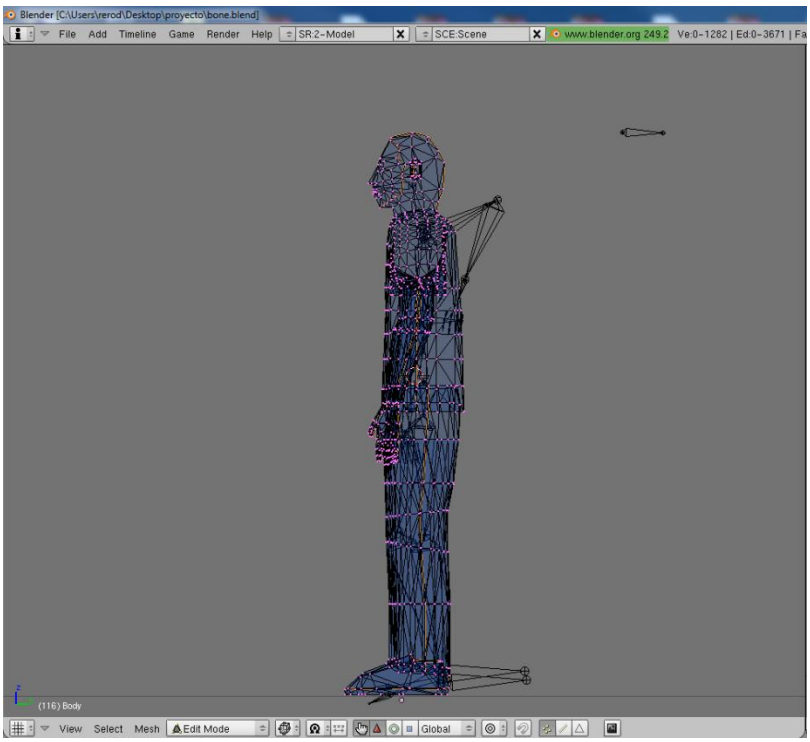

Imagen 124

Ahora pulsamos "F9", y en el área inferior nos aparecerá una caja titulada: "Link and materials" (Imagen 125):

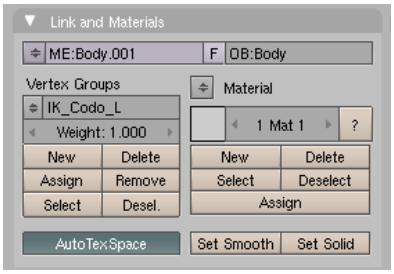

Imagen 125

Y en "Vertex Groups" desplegamos la primera casilla y seleccionamos el nombre del hueso "manipulador" que influye en uno de los vértices del codo. En nuestro caso, este hueso se llama "IK_Codo_L".

Lo seleccionamos (Imagen 126, en este caso ya aparece seleccionado).

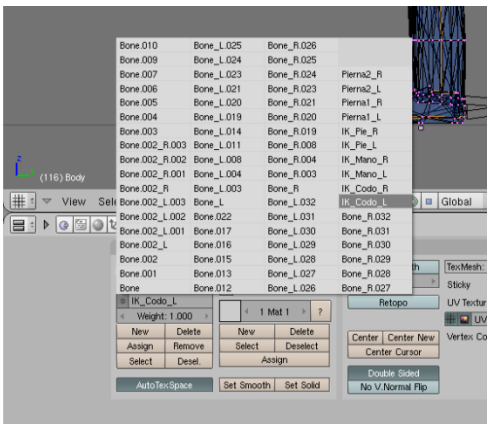

Imagen 126

Una vez seleccionado, pulsamos el botón Select que se encuentra abajo. Y nos aparecerá resaltado en amarillo el vértice que se movía con el "manipulador". Al pulsar el botón "Select", lo que ha hecho Blender es mostrarnos todos los vértices que se ven influenciados por el hueso seleccionado.

Ahora procedemos a desvincular este vértice del "manipulador". Mantenemos el vértice seleccionado y pulsamos el botón Remove. Con esto debería haber desvinculado el vértice. Para comprobarlo, pulsamos "A" sobre la malla, para seleccionar todos los vértices, y volvemos a pulsar "A" para deseleccionar todo. Volvemos a la caja "Vertex Groups", nos aseguramos de que está seleccionado el nombre del hueso: "IK_Codo_L", y pulsamos Select . Si al pulsar este botón no se selecciona ningún vértice en la malla todo ha salido correctamente. Si sigue resaltado el vértice, algo hemos hecho mal, debemos revisar bien el paso anterior. Ahora bien, si hemos conseguido desvincular el vértice, volvemos al "modo pose" y podemos observar que ya no se mueve ningún vértice con el movimiento del "manipulador" (Imagen 127):

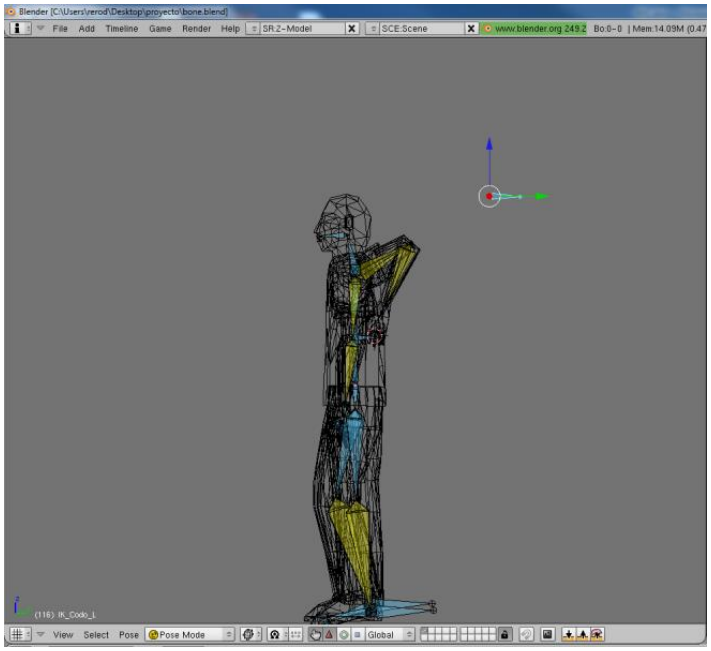

Imagen 127

Debemos aplicar la misma corrección al resto de los vértices mal asignados.

Pasamos a configurar el grado de influencia de cada hueso con los vértices. Entramos en "modo objeto", seleccionamos la malla, y pulsamos "Ctrl+Tabulador" para entrar en modo "Weight paint" (Imagen 128):

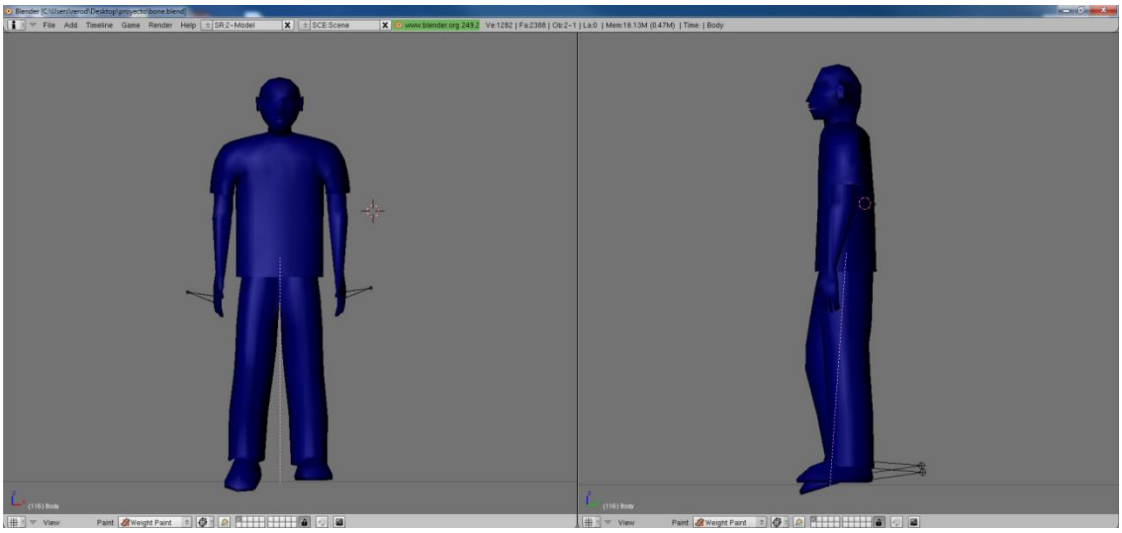

Imagen 128

El color azul indica que el hueso seleccionado, no ejerce ninguna influencia sobre los vértices. Y al contrario, cuanto más se acerque el color al rojo, mayor será la influencia que tiene el hueso sobre esos vértices.

Esta es la distribución de pesos del hueso que forma la base de la columna vertebral (Imagen 129):

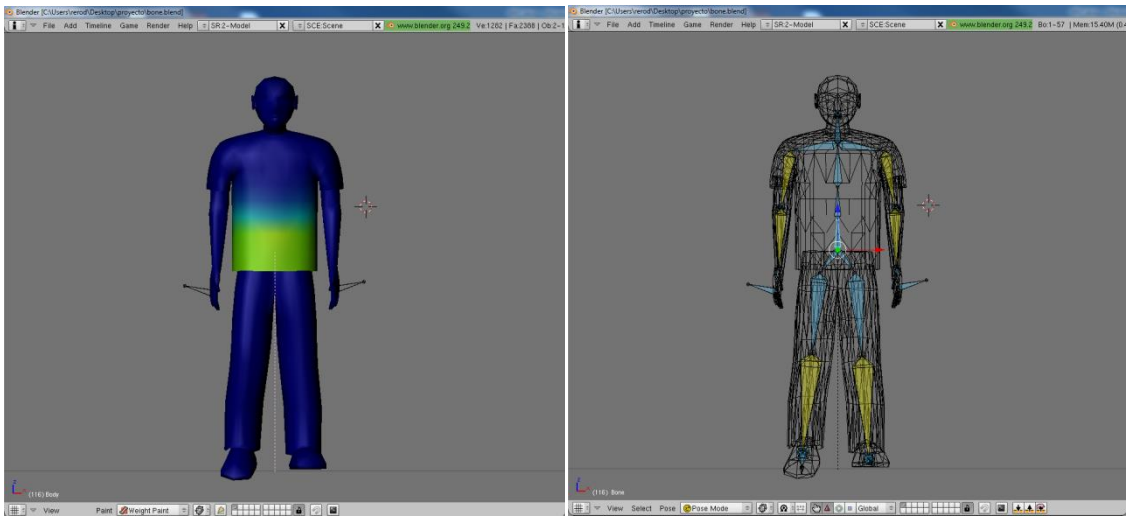

Imagen 129

En la parte izquierda se puede observar que el peso se distribuye uniformemente a lo ancho, y va disminuyendo por arriba (en color azul claro). Sin embargo, el color verde tiene un valor muy alto, por lo que, el movimiento del hueso no se verá muy afectado en estos vértices.

Ahora observamos la distribución de pesos del hueso del muslo izquierdo (Imagen 130):

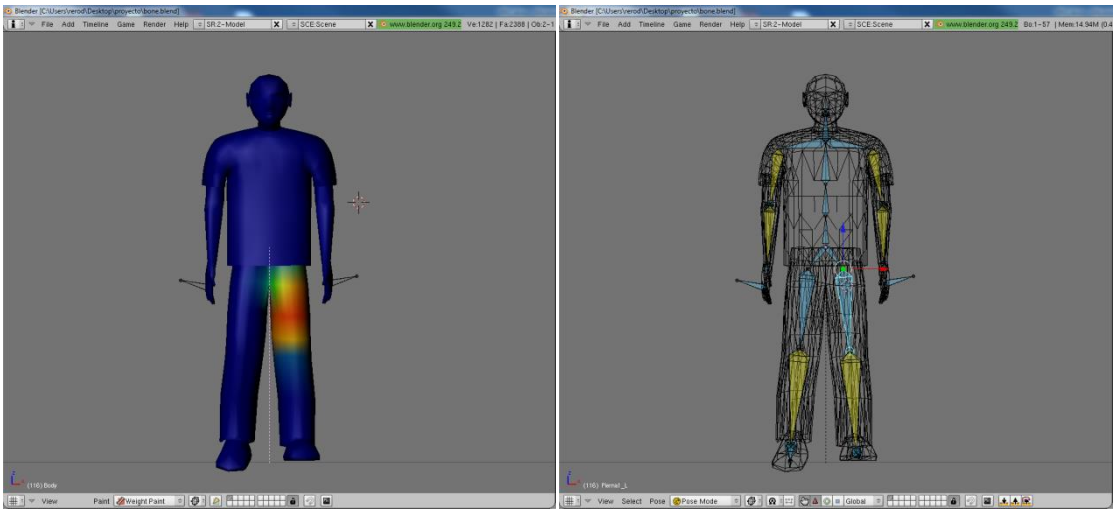

Imagen 130

En esta imagen (Imagen 130), se puede observar que los pesos asignados en los vértices del muslo son más altos. Máximo (en rojo) en el centro del muslo y se va degradando poco a poco al ir acercándose a los dos extremos del hueso.

Esto hace que los vértices se desplacen la misma cantidad que el hueso (Imagen 131):

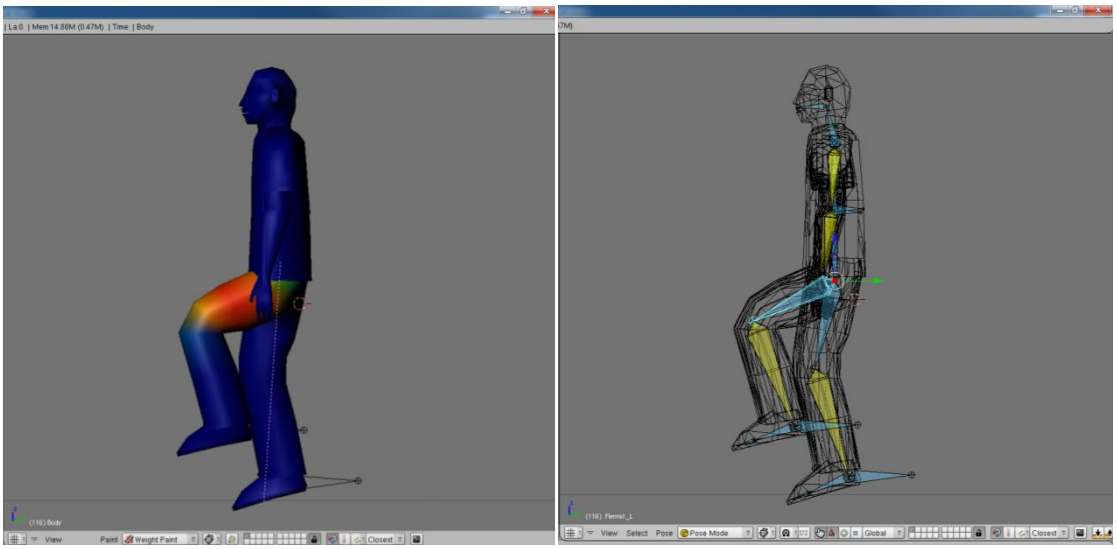

Imagen 131

Si queremos refinar los pesos sólo tenemos que pintar los vértices oportunos con el pincel (haciendo clic con el botón izquierdo del ratón) dentro del modo "Weight Paint". Podemos regular la intensidad de cada clic, en la caja "Paint" cambiando el valor de "Weight" (Imagen 132):

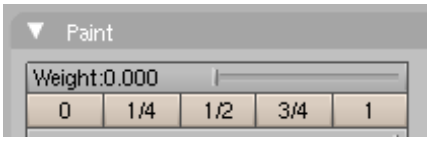

Imagen 132

Y con esto hemos terminado la fase de *rigging*, y por lo tanto el modelo 3D del avatar.

A2.5 ANIMACIÓN

En este apartado vamos a dar movimiento al avatar. Vamos a utilizar el sistema de animación de Blender. Al final exportaremos la animación un formato llamado MD2 para poder cargar en el simulador.

Empezamos cargando nuestro avatar con su esqueleto en Blender, y nos vamos a la pantalla de animación. Para ello desplegamos el primer menú desplegable que se encuentra en la barra superior y seleccionamos "1-Animation" (Imagen 133):

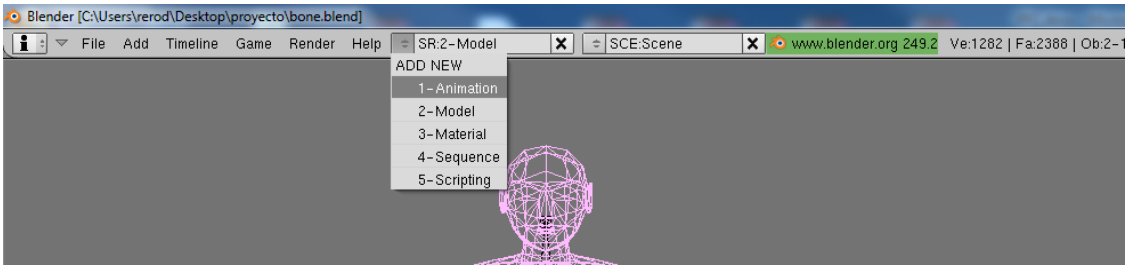

Imagen 133

Y nos aparece la siguiente configuración de pantalla (Imagen 134):

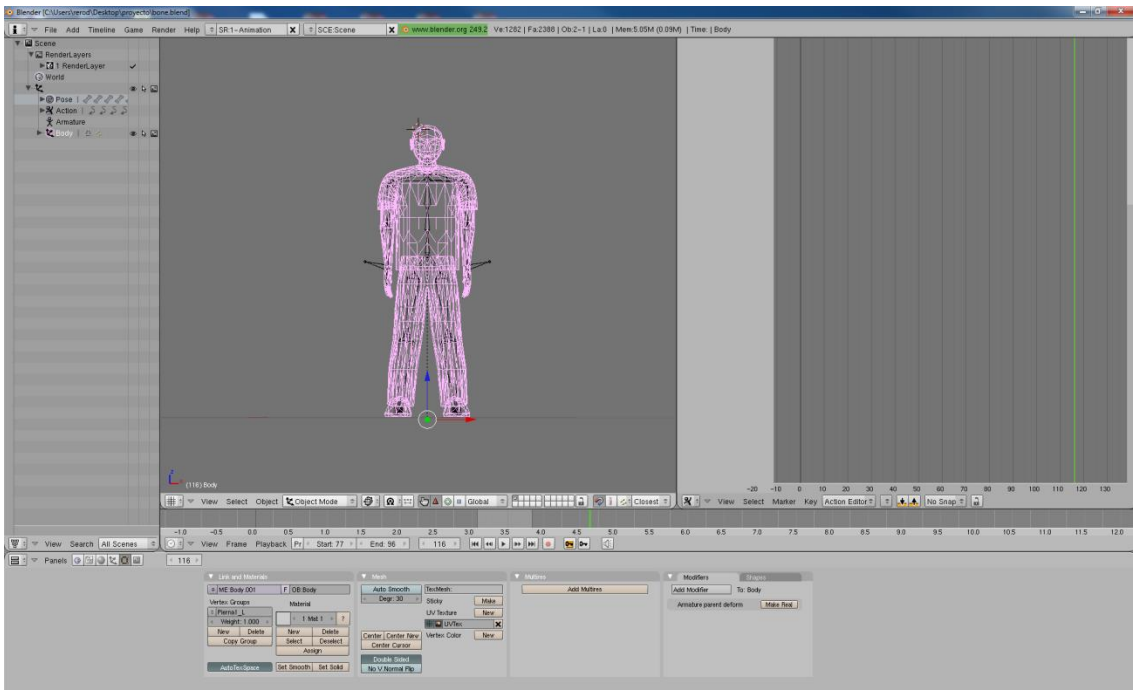

Imagen 134

Una vez preparado el entorno, vamos a comenzar con la creación de la animación. Vamos a crear uno de los movimientos más básicos y necesarios para el avatar: caminar. La técnica que vamos a utilizar para la animación es la animación mediante keyframes. Dado que el movimiento de caminar es cíclico nos bastará con crear sólo un ciclo, y lo reproduciremos con bucles en el simulador.

A partir de aquí vamos a crear una animación que llamaremos "caminar". Para nuestro ciclo de "caminar" utilizaremos cuatro keyframes. Para que sea más simple utilizar la animación en el simulador, todos los keyframes debe tener el avatar en el centro, sin desplazarse en ningún momento del centro de la pantalla. De este modo, cuando empiece a caminar el avatar en el simulador reproduciremos el ciclo de animación y la traslación de la malla la realizaremos desde el simulador. De no ser así se complica el cálculo en el simulador ya que tendríamos que realizar cálculos adicionales para calcular el desplazamiento real del avatar teniendo en cuenta el desplazamiento añadido por la animación.

Para empezar, vamos a definir el número de fotogramas que usaremos para el ciclo completo. Vamos a utilizar 30 fotogramas, lo marcamos en el "Timeline" que aparece justo debajo del área de "3D View" (Imagen 135) y en "Start" ponemos 1 y en "End", 30:

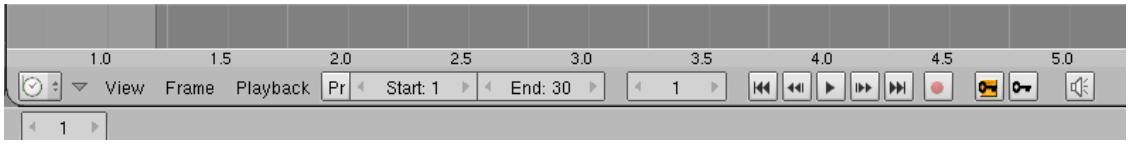

Imagen 135

Una vez hecho esto, pasamos a crear el primer *keyframe*. Movemos los huesos para que el avatar quede con la siguiente pose (Imagen 136):

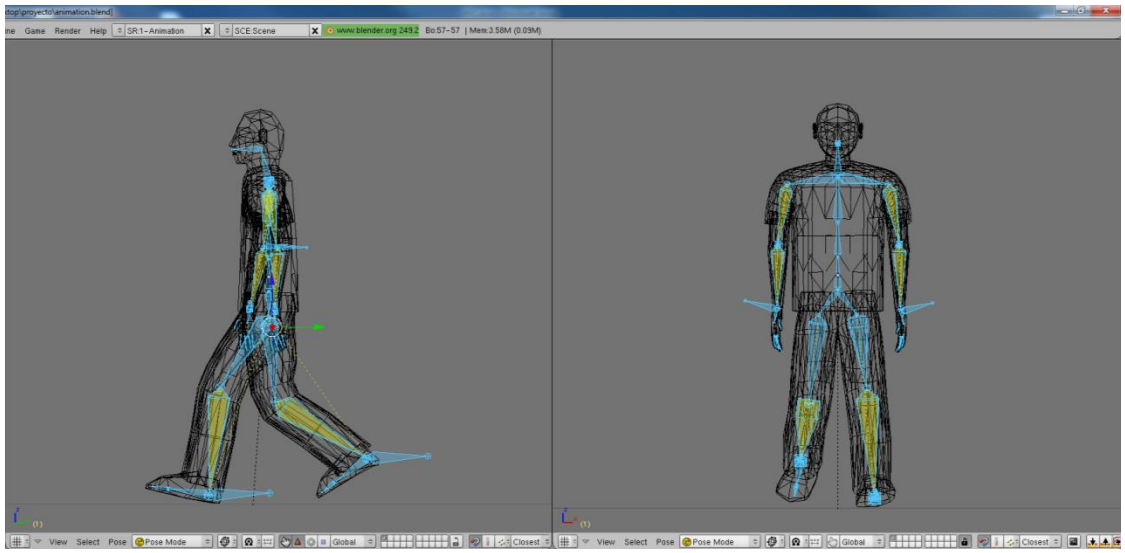

Imagen 136

Ahora seleccionamos los seis huesos "manipuladores" y pulsamos "I" y seleccionamos "LocRot" (Imagen 137):

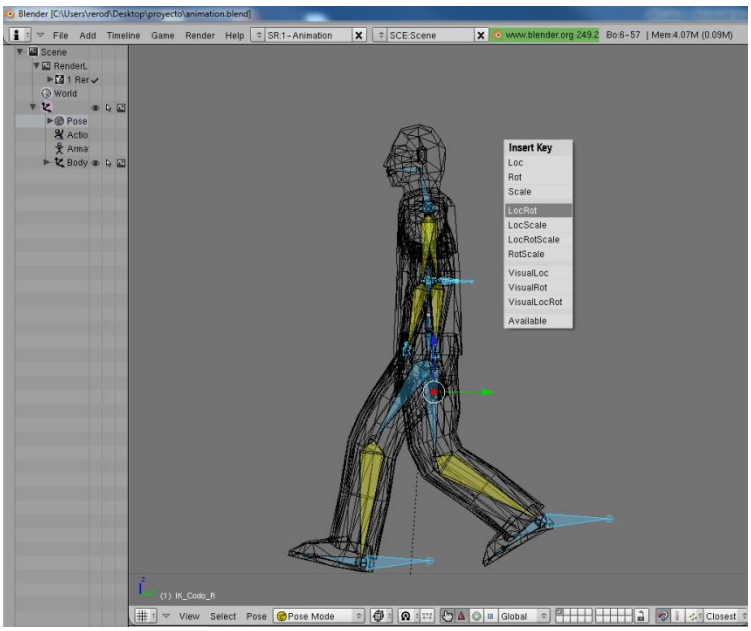

Imagen 137

Imagen 138

Con esto queda registrada la posición y la rotación de los seis huesos en el primer *keyframe* (Imagen 138). Una vez definido el primer *keyframe* vamos a copiarlo en el último frame de la animación para que el ciclo se complete, es decir, que empiece y termine con la misma pose. Para ello, pulsamos el botón  para copiar la posición y la rotación de los seis huesos seleccionados, y movemos el frame de la animación a la posición 31, haciendo clic en la línea vertical etiquetada con el número 31 dentro del "Action Editor" (Imagen 139):

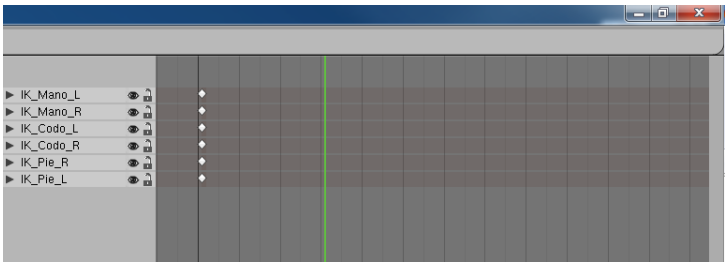

Imagen 139

Y pulsamos el botón  para copiar la posición y la rotación de los seis huesos que hemos copiado previamente. Ahora registramos la pose en el frame 31 pulsando "I" y seleccionando "LocRot". Y queda el siguiente resultado (Imagen 140):

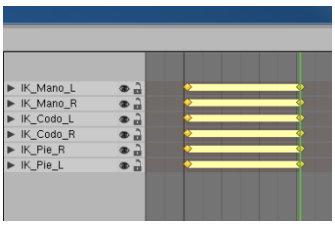

Imagen 140

Con esto tenemos creados el primer y el último *keyframe* de la animación.

El siguiente paso es crear el *keyframe* del medio, que es justo simétrico a la primera pose. Como ya tenemos copiado la primera pose, vamos directamente al frame que corresponde a la mitad del ciclo de animación, en nuestro caso el 16. Y pulsamos el botón  para pegar las propiedades de los seis huesos, pero de forma especular, es decir, las propiedades del hueso izquierdo se asignan a su hueso simétrico del lado derecho, y las propiedades del hueso izquierdo a su hueso derecho. De tal forma que la pose resultante (Imagen 142) queda como si estuviese reflejado en un espejo la pose original (Imagen 141):

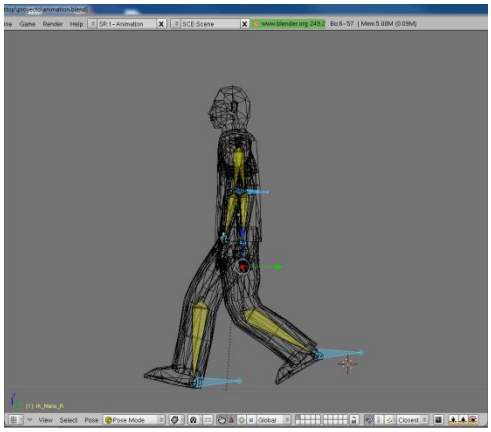

Imagen 141: pose original

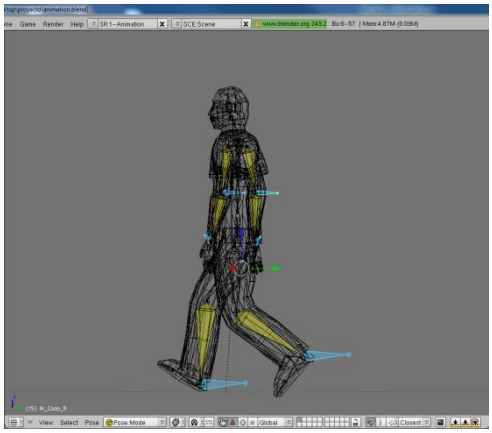

Imagen 142: pose resultante

Lo registramos pulsando "I" y seleccionando "LocRot", y ya tenemos otro *keyframe* más.

Ahora nos situamos en el frame 8 y manipulamos los huesos hasta que nos quede la siguiente pose y registramos el nuevo *keyframe* (Imagen 143):

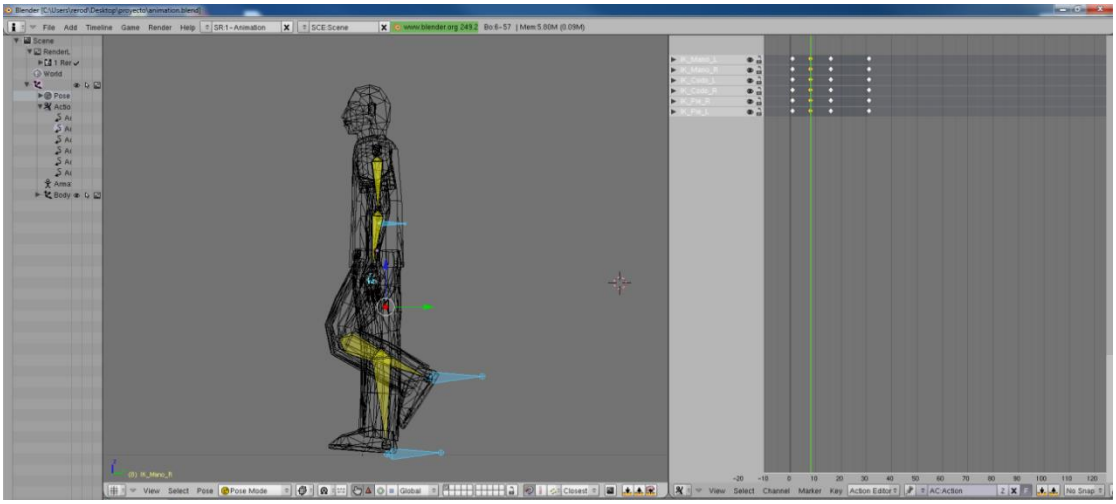

Imagen 143

Copiamos las propiedades de los seis manipuladores de este frame y lo pegamos en el frame 24 y en espejo como hemos hecho anteriormente con el frame 16 (Imagen 144):

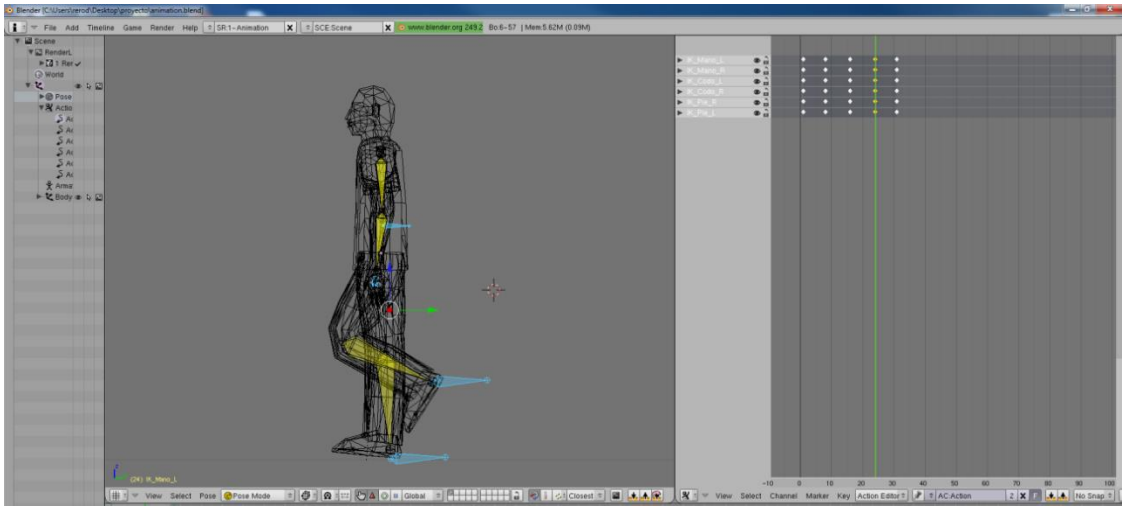

Imagen 144

Con esto tenemos completo el ciclo de "caminar". Pulsamos el botón *play* del "timeline" y se pondrá en marcha la animación que acabamos de crear. El avatar empieza a caminar y completa el ciclo correctamente. Sin embargo, el movimiento no es muy natural, sólo se mueven los pies y los brazos. Vamos a añadir el movimiento del tronco para que parezca más natural. (Otra mejora que podemos añadir es eliminar la simetría perfecta entre los brazos y los pies para que parezca aún más natural.)

Para mover el tronco nos situamos en el frame 8 y seleccionamos todo el esqueleto (Imagen 145):

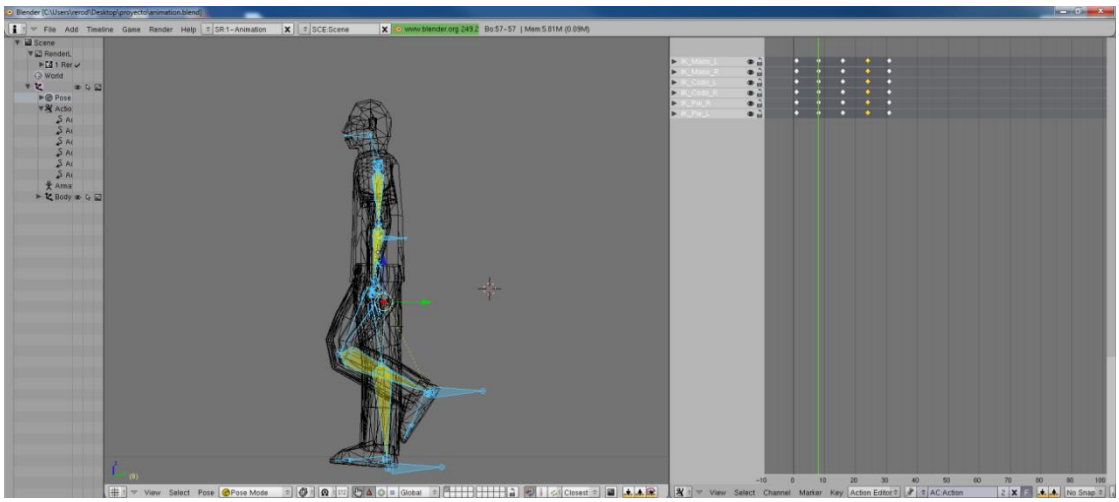

Imagen 145

y pulsamos "G" para desplazar y "Z" para que sólo se mueva sobre el eje Z. Subimos el avatar hacia arriba hasta que su planta del pie se alinee con la línea del suelo (Imagen 146):

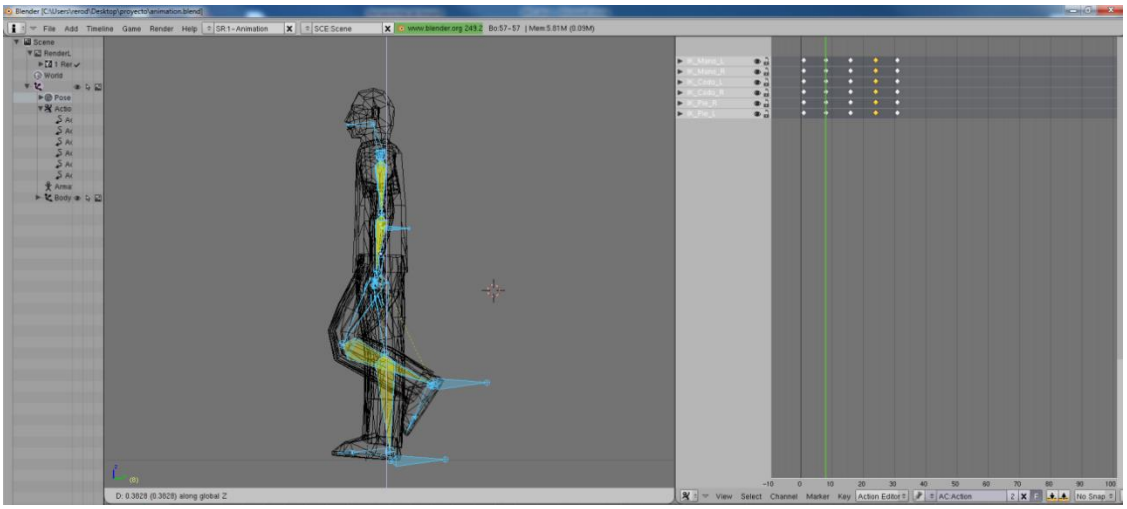

Imagen 146

Seleccionamos los cinco huesos del tronco (tres de la columna y los dos de la cadera) y lo registramos en el frame 8 (Imagen 147):

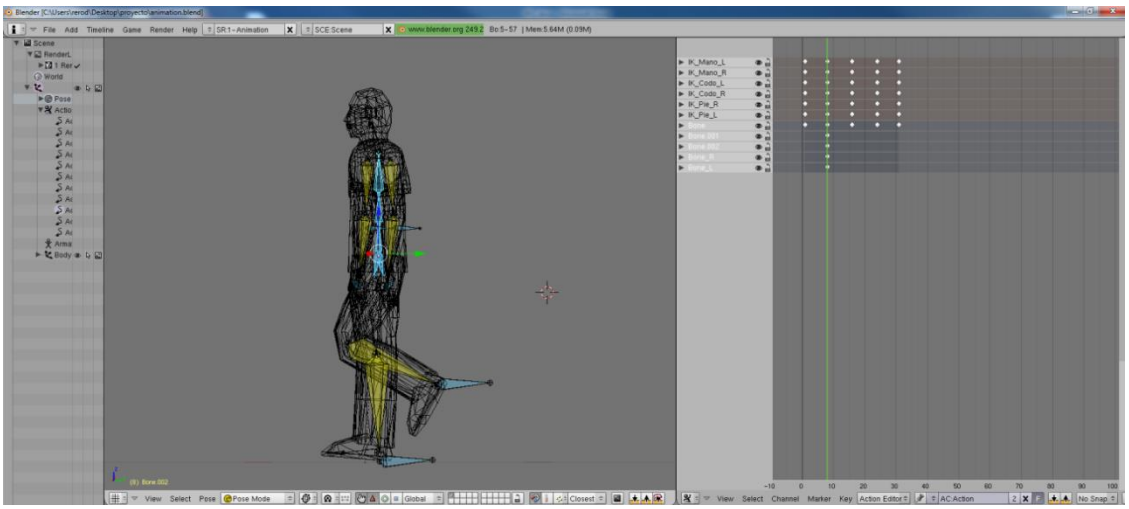

Imagen 147

Ahora vamos al primer frame y bajamos un poco la posición de los cinco huesos seleccionados anteriormente, y lo registramos en este frame (Imagen 148):

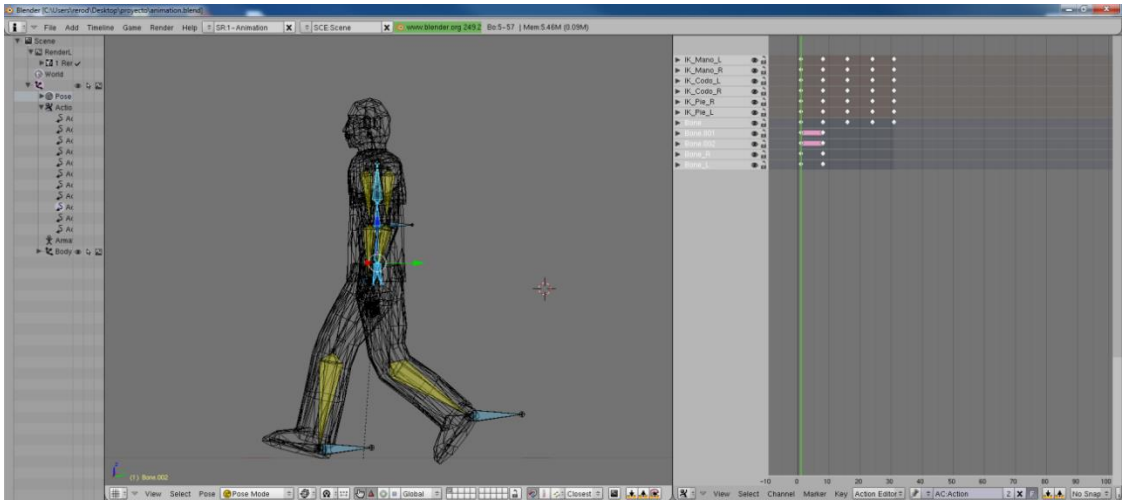

Imagen 148

Antes de terminar, copiamos este frame en el último. En el frame 16 copiamos el primer frame con espejo. Y en el frame 24 copiamos en espejo el frame 8. Quedando el siguiente resultado (Imagen 149):

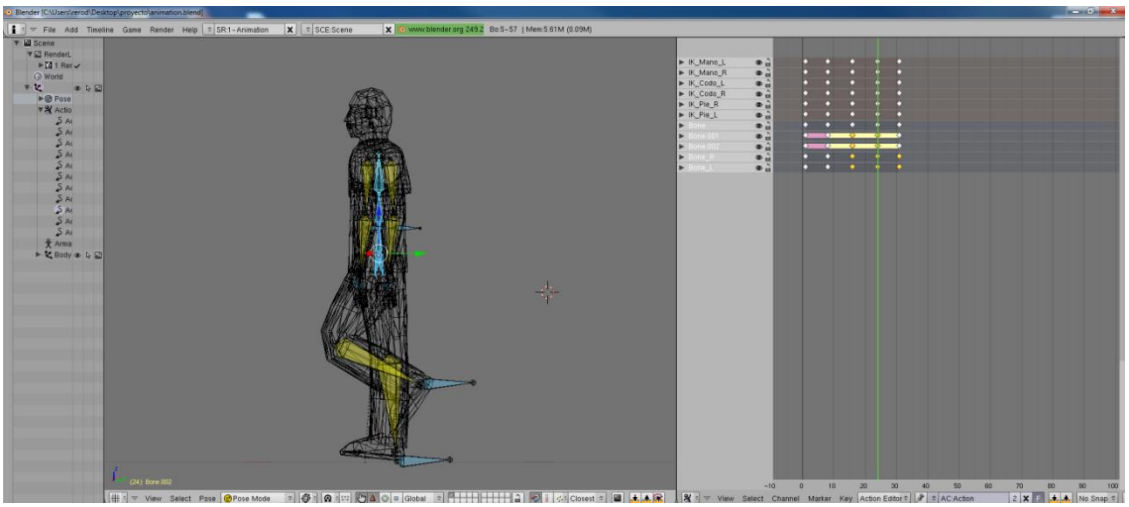

Imagen 149

Y para terminar, vamos a eliminar el frame 31 porque hemos dicho que el ciclo dura 30 fotogramas. Entonces ¿por qué hemos creado el frame 31? La respuesta es muy sencilla: para que el movimiento sea fluido cuando salta del frame 30 al 1. Dado que el frame 31 es igual que el 1, el estado de los huesos en el frame 30 corresponde al frame anterior al 1. Por eso podemos eliminar el frame 31. Pero antes, tenemos que crear el último *keyframe* para que se pueda calcular la interpolación entre el frame 24 y el 30. Bien pues, nos situamos en el frame 30, y el sistema de animación de Blender nos calcula, mediante interpolación, el estado de los huesos en el frame 30 utilizando la información del *keyframe* 24 y el 31 (Imagen 150):

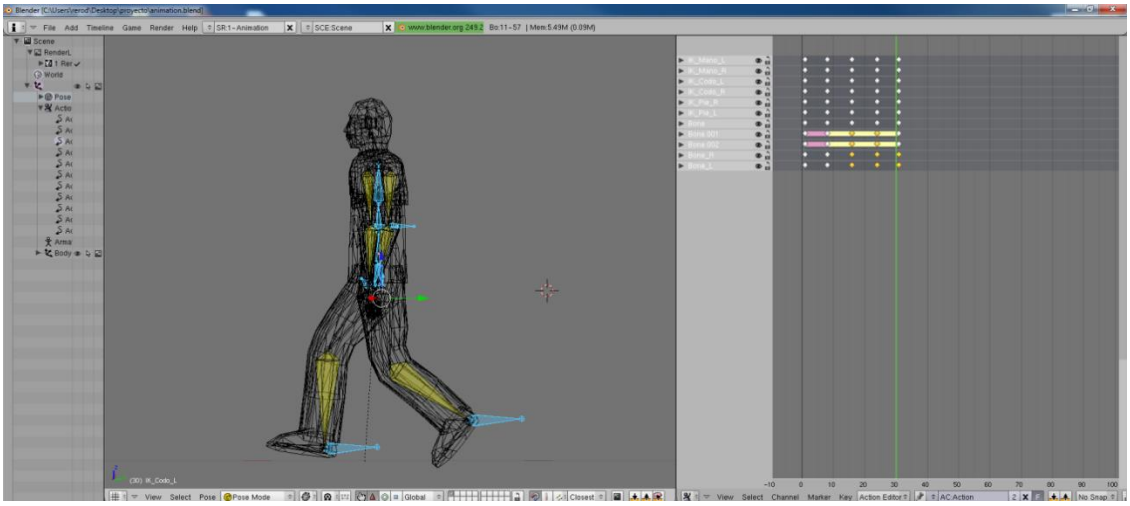

Imagen 150

Ahora seleccionamos los cinco huesos del tronco y los seis huesos manipuladores y registramos el *keyframe* (Imagen 151):

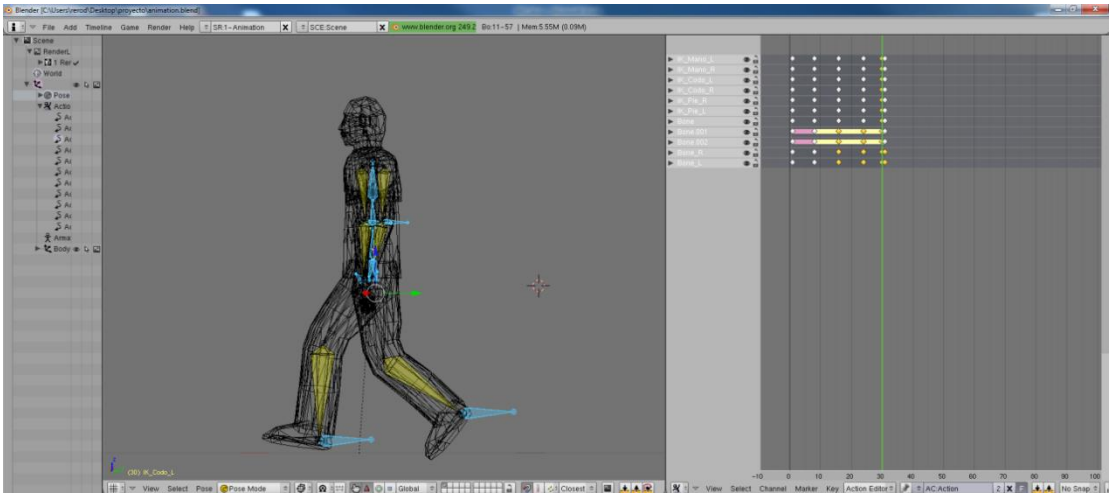

Imagen 151

Y eliminamos el frame 31. Deseleccionamos todos los puntos en el "Action editor" (área de la derecha) pulsando "A". A continuación, nos situamos en el frame 31 y pulsamos "Ctrl+K" para seleccionar todos los puntos del frame y pulsamos "X" y seleccionamos la opción "Erase selected?" (Imagen 152):

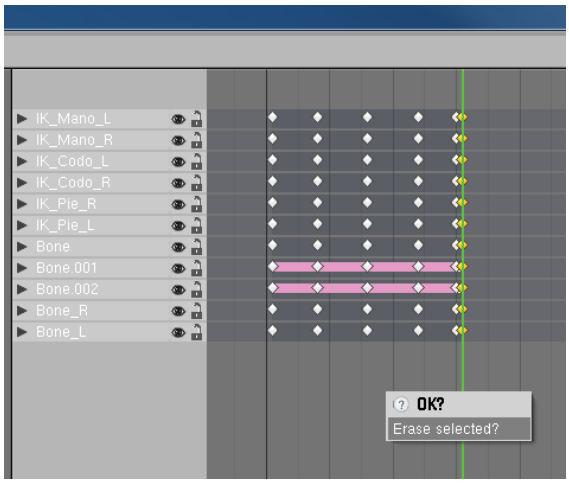

Imagen 152

Y ya tenemos la animación completa (Imagen 153):

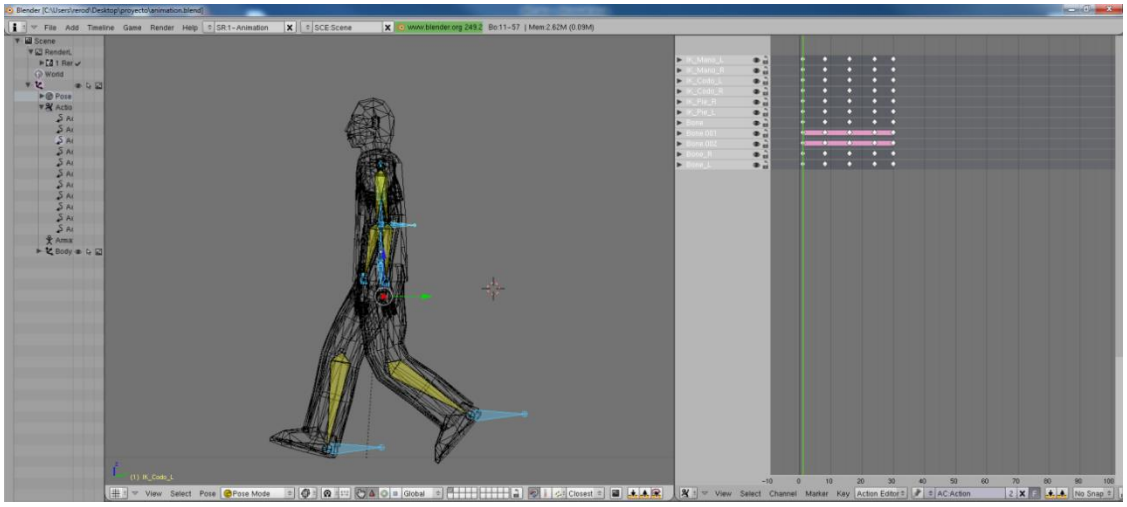

Imagen 153

Con esto hemos conseguido reproducir el movimiento de sube y baja del tronco eliminando la sensación de artificialidad que daba antes. Para añadir más detalle en los movimientos repetimos la operación de: seleccionar el hueso a mover, colocarlo en la posición deseada en el frame adecuado, y registrarlo.

A2.6 EXPORTACIÓN DEL MODELO AL FORMATO MD2

Cuando tengamos la animación finalizada la exportamos a un formato llamado MD2, en el cual se almacena toda la información de la malla y los keyframes de la animación. Este formato es muy utilizado por los principiantes en la creación de juegos en 3D dada su sencillez, lo soporta una gran variedad de motores gráficos.

Pues bien, vamos a exportar la animación. Para exportar a md2 desde Blender debemos utilizar un script Python que ya viene preinstalado. Primero de todo debemos seleccionar la malla del avatar, y vamos al menú "File" -> "Export" y seleccionamos "MD2 (.md2)" (Imagen 154):

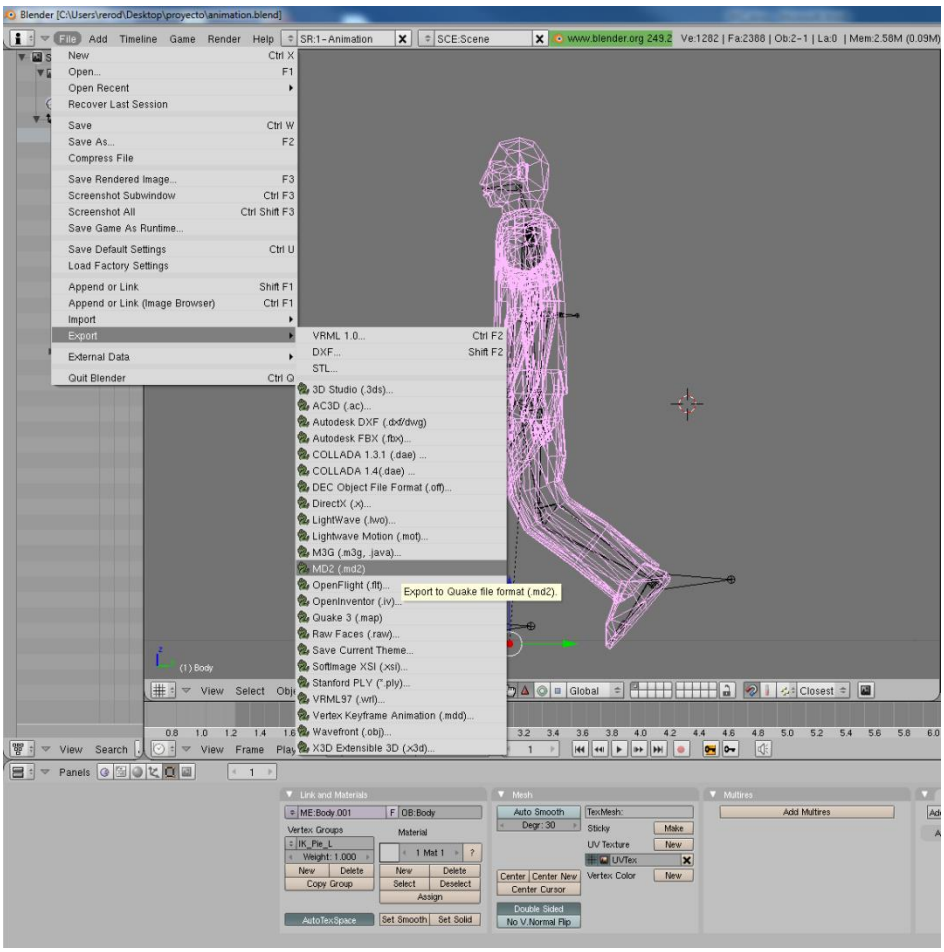

Imagen 154

Nos aparecerá en el área inferior las siguientes opciones para exportar a MD2 (Imagen 155):

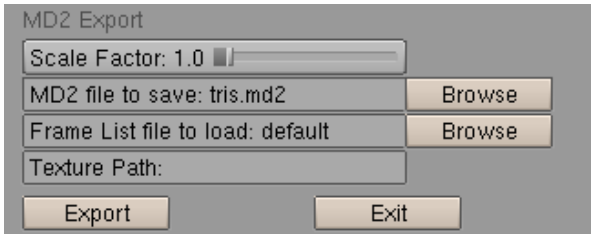

Imagen 155

En "Scale Factor" seleccionamos la escala del avatar en el fichero de salida, en este caso: 1, porque no queremos cambiar su tamaño original.

En "MD2 file to save:" introducimos el nombre del fichero de salida.

En "Frame List file to load" cargamos un fichero de texto que contenga la lista de animaciones con el número de frames que ocupa cada una de ellas. En nuestro caso sólo tenemos una animación: "caminar" que ocupan 30 frames. Por lo que necesitamos crear el siguiente fichero en el bloc de notas (o cualquier otro editor de texto) y cargarlo (Imagen 156):

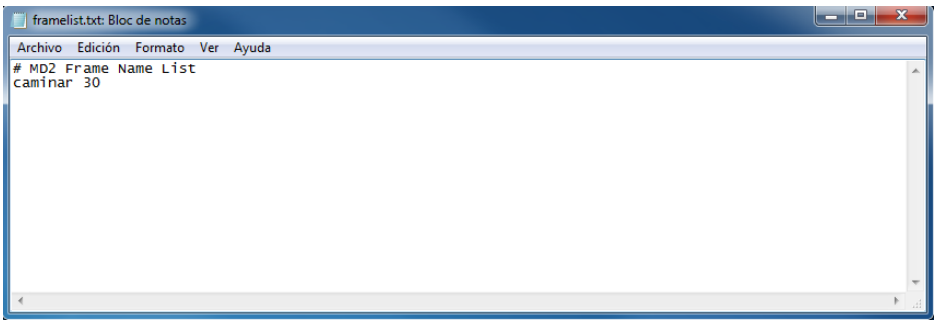

Imagen 156

Es importante que la primera línea del fichero contenga el texto "# MD2 Frame Name List", de no ser así el script de exportación ignorará nuestro fichero. Si queremos exportar más animaciones debemos añadir una nueva línea indicando el nombre de la animación y el número de frames que ocupa.

Y la última casilla "Texture Path" la dejamos vacía.

Así quedarían las opciones de exportación (Imagen 157):

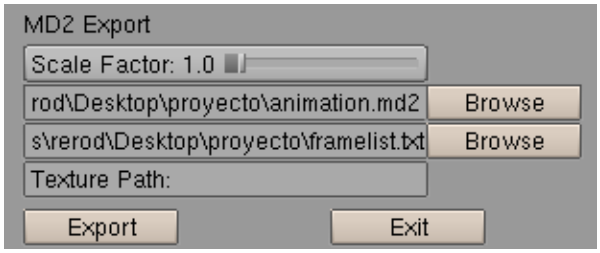

Imagen 157

Para terminar, pulsamos el botón "Export", esperamos unos segundos, y ya tenemos el fichero md2 exportado (Imagen 158):

Imagen 158